AFRL-IF-RS-TR-2006-244
Final Technical Report
July 2006

PROCESSING-IN-MEMORY TECHNOLOGY FOR KNOWLEDGE DISCOVERY ALGORITHMS

University of Southern California

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2006-244 has been reviewed and is approved for publication

APPROVED: /s/

CHRISTOPHER J. FLYNN
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JUL 06	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Dec 04 – Nov 05

4. TITLE AND SUBTITLE PROCESSING-IN-MEMORY TECHNOLOGY FOR KNOWLEDGE DISCOVERY ALGORITHMS	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-04-1-0265
	5c. PROGRAM ELEMENT NUMBER N/A
6. AUTHOR(S) Mary Hall, Hans Chalupsky, Jacqueline Chame, Jafar Adibi and Tim Barrett	**5d. PROJECT NUMBER** HPCS
	5e. TASK NUMBER 31
	5f. WORK UNIT NUMBER 20
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California Information Sciences Institute 4676 Admiralty Way, Suite 1001 Los Angeles CA 90089-8001	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFTC 525 Brooks Rd Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-244

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 06-499

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The goal of this project was to gain insight into whether processing-in-memory (PIM) technology can be used to accelerate the performance of link discovery (LD) algorithms, which represent an important class of emerging knowledge discovery techniques being used by DoD to identify complex, multi-relational patterns. To this end, we evaluate the mapping of LD algorithms to a PIM workstation-class architecture, the DIVA/Godiva hardware testbeds developed by USC/ISI. These hardware testbeds incorporate PIMs into the memory of a conventional processor. Our performance measurements on bandwidth benchmarks, StreamAdd and RandomAccess, show that our prototype PIMs offer increased memory bandwidth to the applications over the Itanium2, with a commensurate increase in performance. The raw performance measurements for two LD kernels show a slowdown on a single PIM, but our analysis shows a performance gain when the differences in clock speed and data scaling are taken into account.

15. SUBJECT TERMS
Processing-In-Memory, Link Discovery, High Performance Computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Christopher J. Flynn
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	46	**19b. TELEPONE NUMBER (*Include area code*)** N/A

Abstract

The goal of this project was to gain insight into whether processing-in-memory technology can be used to accelerate the performance of ***link discovery*** algorithms, which represent an important class of emerging knowledge discovery techniques being used by DOD to identify complex, multi-relational patterns. Such algorithms can greatly benefit law enforcement and intelligence organizations in their efforts to detect and prevent illegal and fraudulent activities as well as threats to national security.

As LD algorithms are data-intensive and highly parallel, involving read-only queries over large data sets, parallel computing power extremely close (physically) to the data has the potential of providing dramatic computing speedups. For this reason, we evaluated the mapping of LD algorithms to a ***processing-in-memory (PIM)*** workstation-class architecture, the DIVA / Godiva hardware testbeds developed by USC/ISI. These hardware testbeds incorporate PIMs into the memory of a conventional processor. The Godiva prototype, the focus of the investigation, includes PIMs as replacement memories in an HP Long's Peak Itanium-2 system.

To test the hypothesis of whether the memory bandwidth provided by PIMs will accelerate link-discovery algorithms, we selected two representative data and graph mining algorithms, condensed them to their core functionality that exhibits most of the computational complexity, and implemented them on the hardware testbed. To compare with a conventional architecture, we also measured the performance of the kernels on the Itanium-2 host. In addition, we performed a similar experiment on two benchmarks that are commonly used for measuring memory bandwidth, RandomAccess and StreamAdd.

Our performance measurements on the bandwidth benchmarks show that our prototype PIMs offer increased memory bandwidth over the Itanium2, with a commensurate increase in performance. Speedups of 8x are shown on the two benchmarks, even though the Itanium-2 has a clock rate 6X faster. The raw performance measurements for the LD kernels show a slowdown on a single PIM, but our analysis shows a performance gain when the differences in clock speed and data scaling are taken into account.

In addition to this performance measurement, we also developed a MATLAB simulator of a parallel LD algorithm, so that we could quickly derive measurements on algorithm variations, and impact of data set properties on performance metrics.

Table of Contents

List of Figures and Tables iii
1. Introduction 1
2. Background 2
  Link Discovery Algorithms 2
  PIM System Environment 3
    Hardware Platform 3
    Compiler Frontend Technology 5
3. Benchmarks 7
  Bandwidth Benchmarks 8
    StreamAdd 8
    RandomAccess 8
  Link Discovery Benchmarks 10
    Mutual Information 10
    Graph Clustering using In/Out Ratios 12
  Parallel Graph Clustering Algorithm 13
4. Parallel Graph Clustering Algorithm Simulation 16
  Analysis of Sample Data Sets 16
  Experimental Setup 19
    Cost 20
    Load Balance 21
5. Performance Analysis on PIMs 25
  Bandwidth Benchmarks 26
    StreamAdd 26
    Random Access 27
  Mutual Information on a Single PIM 29
  Graph Clustering, Sequential and Parallel 31
  Scaling Analysis and Projected Performance 33
6. Summary, Conclusions and Future Directions 35
7. Outreach 36
References 38

List of Figures and Tables

Figure 1. DIVA DIMM with two PIMs. 4
Figure 2. PIM architecture. 4
Figure 3. PIMs installed in HP zx6000 Long’s Peak Itanium-2 system. 5
Figure 4. DIVA PIM Compiler Technology 6
Figure 5. Speedups due to compiler optimizations. 7
Figure 6. Bandwidth Benchmarks 8
Figure 7. Host and PIM collaborating RandomAccess code. 9
Figure 8. Illustration of Host+PIM collaboration on RandomAccess. 9
Figure 9. High-level MutualInformation Algorithm. 11
Figure 10. Graph clustering algorithm using InOut ratio……………………………… 12
Figure 11. Link Table Representation. 13
Figure 12. PIM Parallel In/Out Ratio Algorithm, in MPI 14
Figure 13. PIM reduction computation for parallel IOR algorithm. 15
Figure 14. MPI reduction computation for parallel IOR algorithm. 16
Figure 15: Summary of ARDA/Eagle Data Sets. 17
Figure 16: Representation of 2 data sets, demonstrating their differences. 18
Figure 17: Scatterplot of all data sets. 19
Figure 18: Processing Cost vs. Communication Cost. For 1,2,4,8 and 16 PIMS.1,2,3,4. 20
Figure 19: Load imbalance in 8 PIMs, Group Size = 10, M = 1 (communicate one) 21
Figure 20: Load imbalance: 8 PIMs, Group Size = 10, M = 2 (communicate two) 22
Figure 21: Cost vs. Connectivity 24
Figure 22: Speedup vs. Connectivity 24
Figure 23. Performance Tuning Process on Hardware. 26
Figure 24. StreamAdd Performance Measurements……………………………… .27
Figure 25. G-updates/second for RandomAccess. 28

Table 1. Benchmark Programs……………………………………………………… 6
Table 2: Summary of Load Balance Experiment 23
Table 3: Computation and Communication cost for 1,2,4,8, 16 PIMs for M = 1,2......... 23
Table 4: Result Deterioration (F-value compared to the M =1 case) ………………..…25
Table 5. Performance comparison for MutualInformation on TEST input. 29
Table 6. Mutual Information Data Scaling on Itanium-2 31
Table 7. Performance comparison for Graph Clustering on TEST input. 31
Table 8. Graph Clustering Data Scaling on Itanium-2 32
Table 9. Comparison of Itanium-2 and PIM processors. 33
Table 10. Projections of commercial implementations of PIMs for 2007 and 2010. 34
Table 11. Projected performance accounting for clock and data scaling. 34

1. Introduction

Link discovery (LD) algorithms represent an important class of emerging knowledge discovery techniques being used by DOD to identify complex, multi-relational patterns. Such algorithms can greatly benefit law enforcement and intelligence organizations in their efforts to detect and prevent illegal and fraudulent activities as well as threats to national security. LD algorithms are data-intensive and highly parallel, involving read-only queries over large data sets.

Developing parallel link discovery algorithms presents a variety of difficult challenges. First, data ranges from highly unstructured sources such as reports, news stories, etc. to highly structured sources such as traditional relational databases. Unstructured sources need to be preprocessed first either manually or via natural language extraction methods before they can be used by LD methods. Second, data is complex, multi-relational and contains many mostly irrelevant connections (connectivity curse). Third, data is noisy, incomplete, corrupted and full of unaligned aliases. Finally, relevant data sources are heterogeneous, distributed and can be very high volume.

In this project, we evaluated the mapping of LD algorithms to a ***processing-in-memory (PIM)*** workstation-class architecture, the Data Intensive Architecture (DIVA) [Hall et al, 1999] and Godiva hardware testbeds developed by USC/ISI. These hardware testbeds incorporate PIMs into the memory of a conventional processor. The Godiva prototype, the focus of the investigation, includes PIMs as replacement memories in an HP Long's Peak Itanium-2 system. This testbed was developed under the Godiva project supported by DARPA High Productivity Computing Phase I program, in collaboration with Hewlett-Packard, and completed in February 2004.

We hypothesized that for LD algorithms, parallel computing power extremely close (physically) to the data has the potential of providing dramatic computing speedups. To test whether the memory bandwidth provided by PIMs will accelerate link-discovery algorithms, we selected two representative data and graph mining algorithms, condensed them to their core functionality that exhibits most of the computational complexity, and implemented them on the hardware testbeds. To compare with a conventional architecture, we measured the performance of the kernels on the Itanium-2 host.

At the beginning of this project, the Godiva hardware testbed and associated tools had recently been integrated and demonstrated to work on a single benchmark (NAS CG). No other benchmarks had been tested nor had we tuned the infrastructure to deliver performance measurements. Thus, given the limited timeline of this project, we initially measured performance of two common bandwidth benchmarks (StreamAdd and RandomAccess) which were simpler than the LD algorithms, not just to gather results from these benchmarks but also to develop an approach to performance tuning for this experimental system and tune our tool implementations. PIMs were successful at delivering memory bandwidth comparable to the Itanium-2 with just a single PIM, and 8X more bandwidth with 8 PIMs, in spite of the Itanium-2's far more powerful and 6X faster processor.

In addition to the raw performance measurements of the LD benchmarks, through appropriate scaling and modeling to reflect current and future technology we derive an expected performance gain of 1.3X to 2.6X for a single PIM on the LD benchmarks, and predict speedups of as much as 10X for 8 PIMs.

Beyond the performance measurements on the PIM, the project also included two other activities. First, to target the individual PIM processors, the DIVA/Godiva team has developed an innovative front-end compiler technology to exploit the processor's 256-bit datapath automatically from sequential code based on SUIF [Shin et al., 2002a] [Shin et al., 2002b] [Shin et al., 2003], and a back-end technology based on GCC. Additional work on this compiler to deal with larger and more complex application code features, including parallelization of control flow constructs, was completed under this project [Shin et al, 2004] [Shin et al, 2005]. This allowed the compiler to generate SIMD parallel code for the wide datapath for both of the LD kernels.

A second additional activity was an algorithm simulation in MATLAB to test out alternative parallel algorithms and impact of data set properties without having to run a large number of cases on the hardware testbed. These measurements demonstrated that there is potential for PIMs in speeding up graph clustering as the data scales, and particularly for highly connected graphs. We identified load imbalance, and also found that algorithms that drop communication sacrifice precision with just modest performance improvements.

The remainder of this report is organized into seven additional sections. The next section presents background on our previous work on link discovery algorithms and PIM architecture development, which provided the foundation for this project. The subsequent section presents the bandwidth and LD benchmarks. The fourth section describes the MATLAB simulation of alternative parallel LD algorithms. The performance analysis and scaling results are presented in the fifth section, followed by a summary. Two sections at the end of the document describe outreach activities and deliverables.

2. Background

Link Discovery Algorithms

The link discovery codes used in our experiment focus on a specific problem called ***group detection.*** The group detection task can be qualified into either (1) discovering hidden members of *known groups* (or group extension), or (2) identifying completely *unknown groups*. A known group is identified by a given *name* and a set of known members. The problem then is to discover potential additional hidden members of such a group given evidence of communication events, business transactions, familial relationships, etc. In our study, we only focus on *known groups*.

To extend known groups we generally start with a set of known seed members (*e.g.,* a group of suspects) and then proceed in three phases. First, a function θ is applied to find likely new candidates for each group, producing an extended group. Second, the same function θ is used to rank these likely members by how strongly connected they are to the seed members. Third, the ranked extended group is pruned using a threshold to produce the final output. To discover *unknown groups* we need to locate some initial points and treat them as seed members of a virtual group and apply a similar technique to *known groups*. The process of ***Seed Selection*** can use any number of techniques, but we are using seeds provided by the sample data sets discussed in the next section. Once we have a set of seeds, ***Group Detection*** discovers potential additional members of groups represented by each seed.

Group detection looks for entities that are strongly connected with one or more of the seed members. To compute θ we transform the problem space into a graph in which each node represents an entity (such as a person) and each link represents the set of actions (*e.g.,* emails, phone calls etc.) in which the entities are involved. There are several choices for the function θ. For instance we can use ***mutual information (MI)*** between random variables representing individuals' activities which provides a measure of their dependence. To find two strongly connected entities, we aggregate the known links between them and statistically contrast them with connections to other candidates and the general population. This is done by an MI model that exploits evidence such as individuals sharing an attribute (*e.g.*, their address) or being involved in the same activity (*e.g.*, communicating via email). These attributes and actions are represented as random variables, and we measure connection strength by measuring the Mutual Information (MI) between them. If the variables (or entities) are independent, the MI between them is zero. If they are strongly dependent, the MI between them is large. Hence, MI can be used as an indicator whether two entities are strongly connected to each other compared to the rest of the population.

Alternative approaches to MI are conventional social network techniques such as InOutRatio (IOR). In this and previous reports, we refer to this kernel interchangeably as ***graph clustering***. In this method the measure of membership is the ratio between the number of links within the group and the number of links from group members to nodes outside the group.

PIM System Environment

Hardware Platform

The DIVA system architecture was specifically designed to support a smooth migration path for application software by integrating PIMs into conventional systems as seamlessly as possible. DIVA PIMs resemble, at their interfaces, commercial DRAMs, enabling PIM memory to be accessed by host software either as smart-memory co-processors or as conventional memory. In Figure 1 below, we show 2 DIVA PIMs on a standard memory DIMM board. These PIMs are connected to a host processor through *nearl*y conventional memory control logic. A separate memory-to-memory interconnect enables communication between memories without involving the host processor.

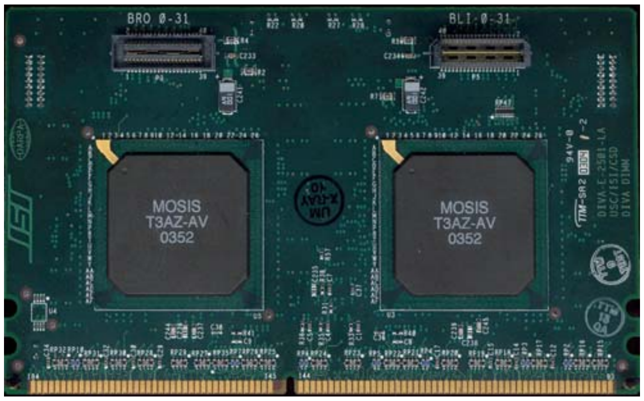


Figure 1. DIVA DIMM with two PIMs.

Aside from memory bus accesses the host processor communicates with the PIMs using parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received.

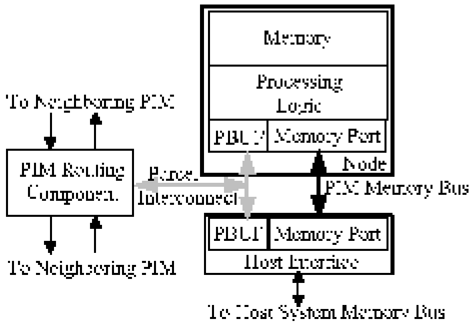


Figure 2. PIM architecture.

Figure 2 shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PIM Routing Component (PiRC). Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. The host interface also contains a parcel buffer (PBUF) for parcel communication between host and PIM. Each PIM node also has a PBUF, for node-to-node parcel communication.

The DIVA PIM node processing logic supports single-issue, in-order execution, with 32 bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath that performs operations similar to those of standard 32-bit integer units, and a 256-bit Wide-Word datapath that performs fine-grain parallel operations on 8-, 16-, or 32-bit operands. The Wide-Word datapath is similar to multimedia extensions like AltiVec. Both datapaths execute from a single instruction stream under the direction of a single 5-stage pipeline, complete with register forwarding logic to resolve data dependence hazards. This pipeline fetches instructions from a small instruction cache, which is included to minimize memory contention between instruction reads and data accesses. Each datapath has its own independent general-purpose register file with 32 registers. Special instructions permit direct transfers between register files without going through memory.

The PIM devices used in the experiments represent our second-generation devices. As compared to the first-generation devices, discussed in [Draper2002], the Godiva PIMs include address translation and eight parallel floating-point units for concurrent use on the Wide datapath. In addition, their memory implements a Double-Data-Rate (DDRAM) interface for integration into an Itanium-2. Figure 3 is a picture of the PIMs inserted in an HP Long's Peak (zx6000) Itanium-2 system.

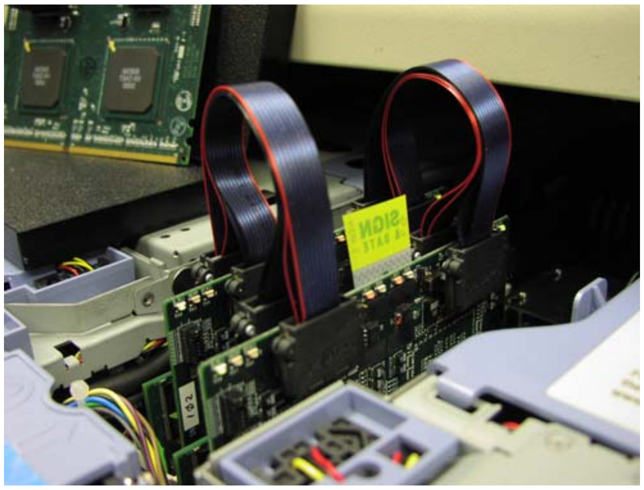


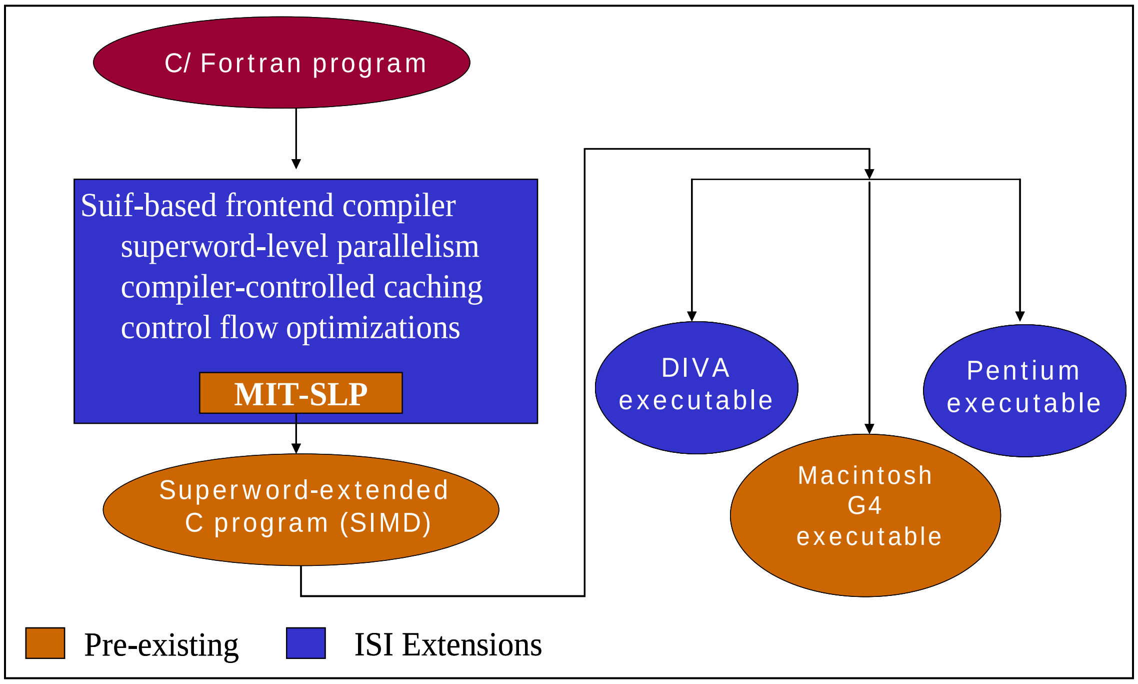
Figure 3. PIMs installed in HP zx6000 Long's Peak Itanium-2 system.

Compiler Frontend Technology

In prior work, we developed a compiler for the PIM processor that generates optimized code in the PIM ISA [Shin 2002a][Shin 2002b][Shin 2003]. Figure 4 illustrates the components of the DIVA compiler. The DIVA front-end compiler is based on SUIF, a research compiler infrastructure developed at Stanford University. The SUIF-based DIVA front end takes as input a C or Fortran program and generates optimized Single Instruction Multiple Data (SIMD) code, a C representation with extensions for superword-level parallelism (SLP) developed for the PowerPC AltiVec. The optimized SIMD code is the input to the DIVA compiler backend. The DIVA backend is based on the Gnu GCC 2.95.3 compiler, ported from the PowerPC toolset. GCC is a commonly used optimizing compiler, but it targets conventional scalar instruction sets. To support optimizations targeting the unique bandwidth-exploiting features of the DIVA ISA, we developed front-end compiler technology that performs DIVA-specific optimizations:

- *superword-level parallelism:* The PIMs' Wide functional unit has operations similar to a multimedia extension architecture such as the PowerPC AltiVec, where the data type is larger than a machine word, and can be configured to perform SIMD parallel operations on different field widths, 8-, 16- and 32-bit.
- *compiler-controlled caching:* The PIMs do not have a data cache, so it is desirable for the compiler to cache reused data in the register file associated with the wide datapath, representing 1KB of storage very close to the processor.

- *control flow optimizations:* As part of the current effort, we have extended this compiler to perform SIMD parallelization in the presence of control flow.



C/ Fortran program
Suif-based frontend compiler
superword-level parallelism
compiler-controlled caching
control flow optimizations
MIT-SLP
Superword-extended C program (SIMD)
DIVA executable
Macintosh G4 executable
Pentium executable
Pre-existing
ISI Extensions

Figure 4. DIVA PIM Compiler Technology

Figure 5 shows the impact of this compiler's optimizations on performance, as measured by speedup over sequential performance. Table 1 describes the benchmarks used in this experimental performance evaluation.

Name	Source	Description	Input Size
VMM	multimedia kernel	vector-matrix multiply	512 elements
FIR	multimedia kernel	finite impulse response filter	256 filter, 1M signal
YUV	multimedia kernel	RGB to YUV conversion	32 K elements
MMM	multimedia kernel	matrix-matrix multiply	512 elements
Swim	SPECfp	shallow water model	SPECfp reference input
tomcatv	SPECfp	mesh generation	SPECfp reference input
Chroma	multimedia kernel	chroma keying	48x48 color image (12 KB)
Sobel	multimedia kernel	Sobel edge detection	1024x768 image (3 MB)
TM	Sandia Labs	image correlation	64x64, 72 32x32 (1.4 MB)
Max	multimedia kernel	max value search	2 100x256x256 (52 MB)
Transitive Closure	DIS Stressmarks	shortest path search	2 1024x1024 nodes (8 MB)
MPEG-dist1	UCLA media bench	dist1 of MPEG-2 encoder	data for first 1000 calls (11 MB)
EPIC-unquantize	UCLA media bench	unquantize_image of unepic	reference input (393 KB)
GSM-calculation	UCLA media bench	full rate speech transcoding	reference input

Table 2. Benchmark Programs

The experiments were performed on the PowerPC AltiVec, as there was not sufficient scope on this effort to port all of these applications to the experimental hardware testbed. Figure 5 shows speedup over a sequential baseline. Our compiler platform extends an earlier SUIF-based compiler from MIT called MIT-SLP. Thus the first (red) bar in the graph shows performance of the MIT-SLP compiler relative to sequential performance. The second (yellow) bar shows our SUIF-based SLP compiler, which extends MIT-SLP to optimize in the presence of constructs such as type casting and reductions. The third (blue) bar shows the control flow optimizations partially developed under this effort [Shin 2004][Shin 2005]. The fourth (green) bar also includes the compiler-controlled caching optimizations.

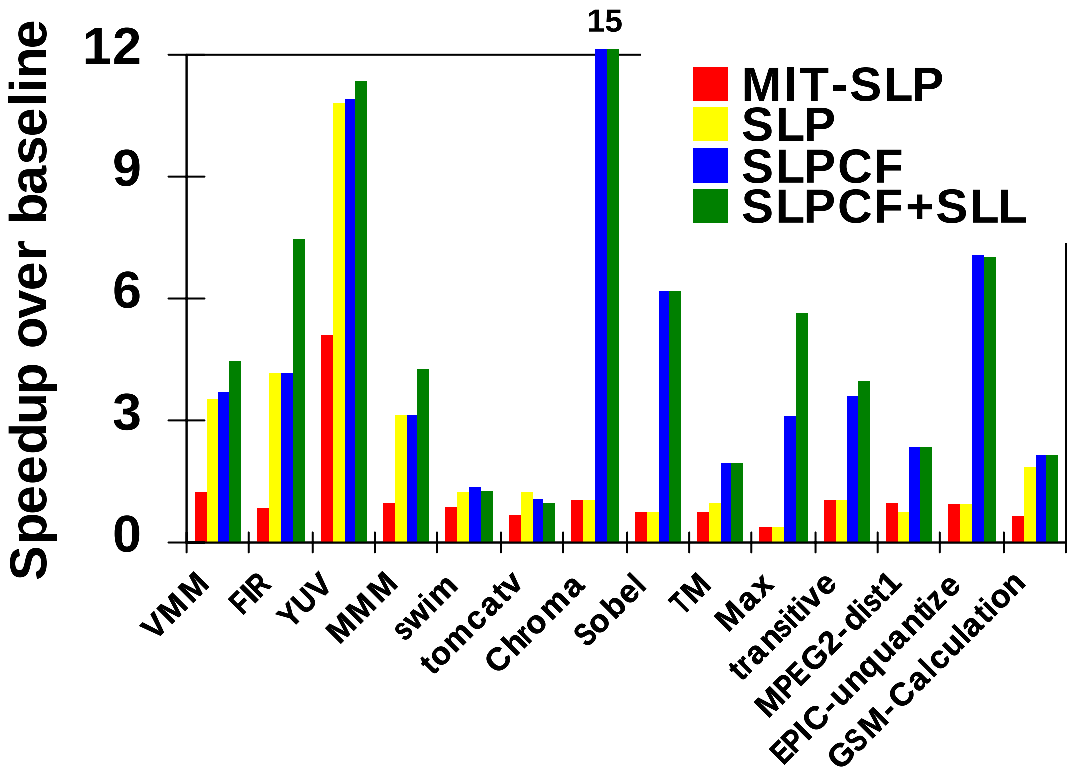


Figure 5. Speedups due to compiler optimizations.

Overall the results show speedups over sequential ranging from 1.05x to 15x for the 14 benchmarks. The control flow optimizations funded by this effort make the difference between little or no speedup and speedups ranging from 1.38x to 15x. We will see in Section 5 that this kind of optimization benefits the MI calculation.

3. Benchmarks

Prior to the experiments with the LD kernels, we performed measurements on two bandwidth benchmarks, StreamAdd and RandomAccess. We felt the addition of these benchmarks was important, as it tests out the hypothesis that PIMs really do offer a bandwidth advantage over conventional architectures. In addition, since the bandwidth benchmarks are less complex than the LD kernels, it provided an opportunity to test out our infrastructure and timing methodology. This section presents all the benchmarks used in the experiment, with original pseudo code and optimized versions targeting the PIMs.

Bandwidth Benchmarks

```
StreamAdd

float a[], b[], c[];
for (i=0; i< DATASIZE; i++)
        a[i] = b[i]+c[i];
```

(a) Host-only code for StreamAdd.

```
RandomAccess

uint64 Table[TABSIZE], ran;
// initialize main table
For (i=; i< TABSIZE; i++)
      Table[i] = i;
// perform updates
ran = 1
for (i=0; i<NUPDATES; i++)  {
    ran = (ran << 1 ^ (ran < 0) ? POLY: 0);
   Table[ran & (tableSize-1]^= ran;
}
```

(b) Host-only code for RandomAccess

Figure 6. Bandwidth Benchmarks

StreamAdd

StreamAdd measures the performance of a stream of floating point additions and updates, and is shown in Figure 6(a). Because there is no reuse of data values in this computation, and very little computation to hide memory latency, it is a useful benchmark for stressing memory bandwidth of architectures. PIMs are effective at speeding up this benchmark, since with the appropriate data distribution, all computation, reads and writes can be performed locally within a PIM. Since the PIM implementation of StreamAdd is straightforward, it is not shown here.

RandomAccess

RandomAccess is part of the HPC Challenge benchmark suite [HPC 2005] and it is designed to stress the memory system by performing updates to randomly selected entries of a large table. Since the updates are to random memory locations there is effectively no spatial reuse. There is a small amount of temporal reuse because each table entry is updated more than once, but the intervals between updates to the same data are typically too large to result in data locality in caches. By design, the table does not fit in the local cache(s) or memory of a system node: the default table size is half of the total memory in the system, and the number of updates to the table is four times the table size.

Figure 6(b) shows the host-only implementation of RandomAccess. For our Itanium2 architecture most random updates result in a cache read miss followed by a write. As the memory latencies are quite high and there is virtually no computation between memory accesses, eventually the memory system is swamped and the application's performance is limited by the main memory system performance.

An alternative implementation of RandomAccess that takes advantage of the PIM processors is to let the PIMs perform the updates locally. The host processor generates the random addresses and, for each update, sends a parcel to the PIM where the table entry resides. The PIMs check for parcels from the host and, once a parcel is received, perform the update to the desired table entry. Figure 7 shows the host and PIM implementations of the PIM-enhanced version of the benchmark, and an illustration that accompanies the code excerpt is shown in Figure 8.

```
// Host code for Host-and-PIM RandomAccess
// initialize main table
for (i=0; i<TableSize; i++)
    Table[i]=i;
ran = 1;
for (i=0; i<NUpdates; i++) {
    ran = (ran << 1) ^ (ran < 0? POLY:0);
    offset = ran & (TableSize-1);
    parcel.command = UPDATE;
    parcel.payload[0] = ran;
    parcel.payload[1] = offset;
    SendParcel (&Table[offset], parcel);
}
parcel.command = DONE;
for (pim=0; pim<NumPims; pim++)
    SendParcel(&done[pim], parcel);
```

```
// PIM code for Host-and-PIM RandomAccess
done = FALSE;
while (!done) {
    // check for parcels from host processor  (non blocking)
    RecvParcel(hostParcelBuffer, recvStatus, parcel);
    if (recvStatus == TRUE) {
        if (parcel.command == UPDATE) {
            ran = parcel.payload[0];
            offset = parcel.payload[1];
            Table[offset] ^= ran; // local memory access
        }
        else if (parcel.command == DONE) {

            done = TRUE;
        }
    }
}
```

Figure 7. Host and PIM collaborating RandomAccess code.

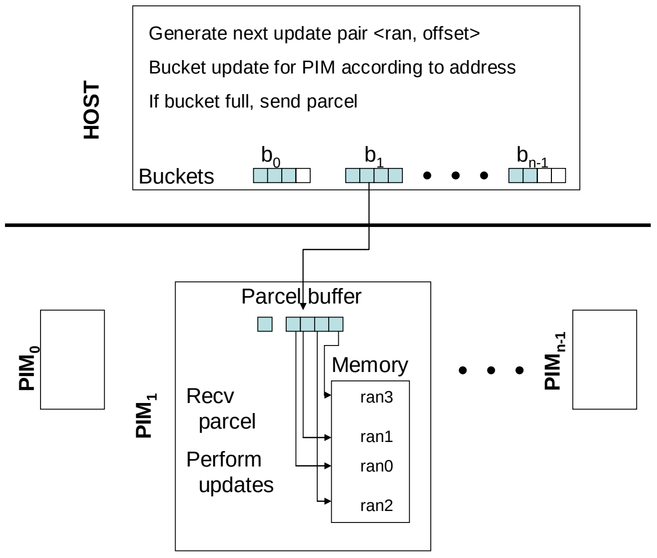


Figure 8. Illustration of Host+PIM collaboration on RandomAccess.

The following feature of our implementation is shown in the illustration of Figure 8 but omitted from the code for simplicity. The host collects four updates destined for each PIM prior to sending a parcel. This is because four <*ran,offset*> pairs can fit in the 256-bit parcel payload. This binning maximizes utilization of the host-to-PIM bandwidth, while also reducing the frequency with which parcels arrive at the PIM for handling.

Link Discovery Benchmarks

Mutual Information

The MutualInformation kernel was initially written in Stella [Stella 2003], a domain-specific language developed by PI Hans Chalupsky for interaction with LD tools. Stella automatically generates C++, and it was the C++ code that was used as the basis for the kernel.

Figure 9 shows a high-level version of the algorithm that computes the mutual information for all node pairs in the graph. For brevity, some operations are shown as "vector" operations akin to Fortran95 array notation, but the actual implementation is in sequential C, and the vector operations on the inner dimension of the arrays are recognized automatically by our compiler.

This computation captures the relationship between edges in a graph, based on conditional probability calculations. For each pair of nodes (representing entities) for which the MI computation is performed, we perform a conditional probability calculation which includes invocation of a log function. For an independent set of pairs, we can perform this calculation in parallel on DIVA's WideWord datapath. There are two large input data structures associated with this algorithm. The first is *LinkTable,* which represents the edges in the graph. There may be several edges in the graph between two nodes, each representing a different type of link *(e.g.,* email, telephone call, etc.). The second is *FrequencyTable,* which represents the frequency of edges between two nodes. The output of this computation is the structure *MiTable*, the result of the mutual information calculation. Other auxiliary structures represent the intermediate results of the mutual information calculation, as will be discussed in Section 5.

The implementation shown in Figure 9 was a result of organizing the computation so that it could be parallelized to support both fine-grain parallelism in the PIM's WideWord unit and coarse-grain parallelism across PIMs. While graph algorithms are thought of as computations with irregular data access patterns, this computation reflects an organization of the graph where some accesses are strided and there are data parallel computations.

```
ComputeAllMi(LinkTable, FrequencyTable, pxv, cpyxt, int nlinks, int ntypes, MiTable)
{
 /* Compute mutual information for all node pairs in  'LinkTable' and deposit results in
'MiTable'.  This computation is O(t^3) where 't' is the number of   link types.  */

   for all indices in LinkTable {

       /* strided access to LinkTable */
        x = LinkTable[index][FROM];
        y = LinkTable[index][TO];
       /* strided accesses on x, irregular accesses on y */
       xfreq = FrequencyTable[x][LINKTYPE];
       yfreq = FrequencyTable[y][LINKTYPE];
       xyfreq = LinkTable[index][1:ntypes];
       /* probability vectors */
       pxv[1:ntypes][link] = FrequencyTable[x][1:nytpes] * (1/nlinks);
       for type = 1, ntypes {
         cpyxt[type][type][link] = yfreq / xfreq;
         cpyxt[0][type][link] = 1.0 – (yfreq / xfreq);
         cpyxt[type][0][link] = (yfreq - xyfreq) / nullfreq;
       }
   }
    aux1[1:ntypes][1:ntypes][1:nlinks] += pxv[1:ntypes][1:nlinks] *
                              cpyxt[1:nytpes][1:ntypes][1:nlinks];
    aux2[1:ntypes][1:ntypes][1:nlinks] +=  MASK(cpyxt[1:ntypes][1:ntypes][1:nlinks]) > 0,
                                            cpyxt[1:ntypes,1:ntypes][1:nlinks] *
                                            LOG(cpyxt[1:ntypes][1:ntypes][1:nlinks] / aux1),
                                            0.0);
    mit[2][1:nlinks] +=  MASK(cpyxt[1:ntypes][1:ntypes][1:nlinks] > 0,
              pxv[1:ntypes][1:nlinks]*cpyxt[j][i][link]*aux2[1:ntypes][1:ntypes][1:nlinks], 0.);
   MiTable[pair][FROM] = x;
   MiTable[pair][TO] = y;
   mi = 0.0;
   aux1 = 0.0
   /* i and j loops for 1 to nytpes are omitted */
   aux2 = 0.0
   for (k = 0; k<ntypes; k= k+1) {
      aux2 += pxv[k] * cpyxt[j][k]; /* i and j are fixed */
       mi = mi + pxv[i] * aux1;
   }
   MiTable[pair][LINK_TYPE] = mi;
   pair = pair + 1;
  }
}
```

Figure 9. High-level MutualInformation Algorithm.

Graph Clustering using In/Out Ratios

As in Mutual Information, a graph *node* represents an entity or individual, and a *link* (or edge) represents one instance of communication between two nodes. Each link has an associated *link type*, which represents the type of communication between the nodes (email, phone call, etc).

The second link discovery kernel performs graph clustering as part of a group finding algorithm. For each node we model its activities with a random variable. The graph clustering component organizes the graph according to the strength of links. This code was initially implemented in MATLAB, and was rewritten in C.

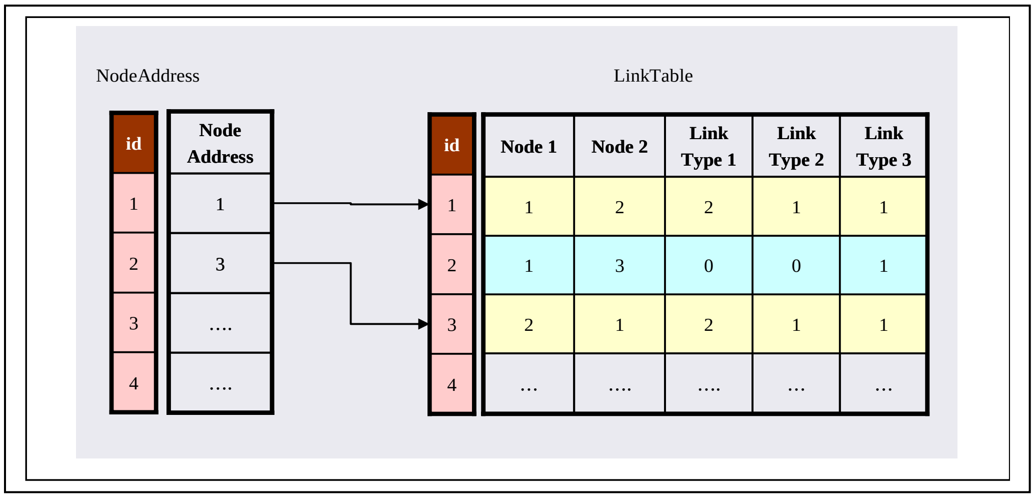
```
algorithm InOutRatio ( {seed members}, numNewMembers) : Group
{
  Group = {seed members}
  bestRatio = 0.0
  while (numNewMembers > 0) {
    foreach node_1 connected to Group {
      newInLinks = newOutLinks = 0
      foreach node_2 connected to node_1 {
        // calculate InOutRatio of node_1
        if (node_2 is in Group)
          newInLinks += ∑(links between node_1, node_2)
        else
          newOutLinks += ∑(links between node_1, node_2)
        InOutRatio=(inlinks+newInLinks)/(outLinks–newInLinks  +  `
                   newOutLinks)
        if (InOutRatio > bestRatio) {
          newMember = node_1
          bestRatio = InOutRatio
        }
      }
    Group = Group U {newMember}
    numNewMembers = numNewMembers - 1
  }
return Group
}
```

Figure 10. Graph clustering algorithm using InOut ratio.

A high-level description of an InOutRatio algorithm is shown in Figure 10. The inputs to the algorithm are the initial cluster and the number of new members to be added to the cluster. Here a *cluster* is a set of nodes such that each node has at least one link to another node in the set. A node in the initial cluster is called a *seed member* and a link among two nodes in the initial cluster is called a *seed link*. A *member* is a node belonging to the cluster.

The algorithm selects new cluster members based on the InOutRatio of nodes that do not belong to the current cluster. At each step, the node with the highest InOutRatio with respect to the current cluster is selected and added to the cluster. This algorithm is a greedy algorithm that only considers one "hop" in connectivity and so the order in which nodes are selected might be an important factor in the final cluster.

The main data structure in InOutRatio is a table that keeps, for each pair of nodes in the graph, the number of links of each link type. Figure 11 illustrates a simple LinkTable. In this example there are three link types, and the first pair of nodes in the table has two links of type 1, one link of type 2 and one link of type 3. A second table (NodeAddress) keeps a pointer to the first pair of each node (that is, the entry of LinkTable corresponding to the first pair of the node). This representation allows the algorithm to compute a node's InOutRatio without having to search the LinkTable.



id	Node Address
1	1
2	3
3	….
4	….

id	Node 1	Node 2	Link Type 1	Link Type 2	Link Type 3
1	1	2	2	1	1
2	1	3	0	0	1
3	2	1	2	1	1
4	…	….	….	…	…

Figure 11. Link Table Representation.

Parallel Graph Clustering Algorithm

We developed a parallel PIM InOutRatio (IOR) algorithm in three steps. First, we implemented the sequential algorithm described above. Then, we implemented a parallel version of the algorithm, described in this section, in MPI (Message Passing Interface) so that we could debug it on a conventional platform. We leveraged a working version of the computation phase of each PIM, which is the same as that of the MPI implementation, and then integrated it with the PIM-to-PIM communication phase, for which we also had prior code that was executing correctly on the hardware. For the PIM version we replaced the MPI constructs with parcels to implement the communication phase of the algorithm, which consists of a parallel reduction. In the MPI version the communication phase is implemented using MPI collective communication functions such as MPI_Bcast and MPI_Reduce. The parallel algorithm is described next.

The data and computation are partitioned among PIM nodes as follows. Each PIM keeps a fraction of the PairTable, that is, a subset of the pairs of nodes in the graph, where a pair is a set of two nodes with at least one link between them. The PairTable is partitioned so that, for a given node, all pairs containing that node are kept on the same

PIM. Hence the whole graph is represented in a link table of *twice the number of all links* times *the number of all link types* plus 2 (for two nodes). This duplication of information avoids unnecessary cross communication among PIMs during the IOR computation. In addition to a subset of the PairTable each PIM keeps a copy of the current cluster in its local memory (initially the current cluster is the set of seed members).

The algorithm iterates until a desired number of new nodes is added to the group, finding a new cluster member at each iteration or until there are no more nodes to be added. At each iteration each PIM computes the node, among the nodes in its subset, with highest InOutRatio. After that, all PIMs communicate to find the node with highest InOutRatio across all PIMs. Finally, at the end of each iteration each PIM adds the new "global" best node to its local copy of the current cluster. Figure 12 shows a high-level version of the PIM InOutRatio algorithm. The parallel reduction, shown in Figure 13, uses parcels to implement the communication among PIMs.

```
algorithm  ParallelInOutRatio ( {seed members}, numNewMembers) : Group
{
Group = {seed members}
LocalOutList = {nodes in local PIM memory connected to Group}
localBestRatio = 0.0
while (numNewMembers > 0) {
    foreach node_1 in LocalOutList {
        newInLinks = newOutLinks = 0
        foreach pair (node_1, node_2) in PairTable {
           if (node_2 is in Group)
              newInLinks += ∑( links of pair (node_1, node_2) )
           else
              newOutLinks += ∑( links of pair (node_1, node_2) )
           InOutRatio = (inlinks+newInLinks)/(outLinks – newInLinks + newOutLinks);
           if (InOutRatio > localBestRatio) {
               localBestRatio = InOutRatio;
               localBestNode = node_1;
           }
        }
    }
  ParallelReduction(localBestNode, localBestRatio, newInLinks, newOutLinks,
                          globalBestNode, inLinks, outLinks)
    Group = Group U {globalBestNode}
    update  LocalOutList
    numNewMembers = numNewMembers  - 1
}
return Group
}
```

Figure 12. PIM Parallel In/Out Ratio Algorithm, in MPI

```
ParallelReduction (localBestNode, localBestRatio, newInLinks, newOutLinks,
                   globalBestNode, inLinks, outLinks)
{
  /* all PIMs compute best node in log(nPims) steps */
  for (step = 0; step < log(nPims); step ++) {
    /* if PIM is a sender at this step */
    if (Sender (myId, step)) {
          dest = ComputeDest(myId, step, nPims)
          SendParcel (dest, localBestNode, localBestRatio, newInLinks, newOutLinks)
    }
    /* else if PIM  is a receiver at this step */
    else if (Receiver (myId, step)) {
      RecvParcel(node, ratio, recvInLinks, recvOutLinks)
      if (ratio > localBestRatio) {
        localBestNode = node
        localBestRatio = ratio
             newInLinks = recvInLinks
             newOutLinks= recvOutLinks
      }
    }
    else { /* neither sender or receiver */
           /* do nothing during this step */
    }
    barrier
  }

  /* PIM0 computes best node and sends it to all */
  if (myId == PIM0) {
        globalBestNode = localBestNode
        inLinks = inLinks + newInLinks
        outLinks = outLinks – newInLinks + newOutLinks
        BroadcastParcel (globalBestNode, inLinks, outLinks)
  }
  barrier

  /* all PIMs update LocalOutList and local copy of InList */
  RecvParcel(globalBestNode, inLinks, outLinks)
  InList = InList U globalBest
  UpdateLocalOutList(globalBestNode)
  barrier

}
```

Figure 13. PIM reduction computation for parallel IOR algorithm.

The computation phase of the MPI version is the same as that of the PIM version. The MPI communication phase implements a parallel reduction using MPI collective communication functions. Figure 14 illustrates a simplified version of the MPI parallel reduction. MPI_Reduce with operator MPI_MAXLOC is used to compute the global maximum ratio and the node corresponding to the maximum ratio. The root process then uses MPI_Bcast to broadcast the global best node to all processes.

```
MPI_ParallelReduction(localBestNode, localBestRatio, globalBestNode)
{
   Local.bestRatio = localBestRatio
   Local.bestNode = localBestNode
   MPI_Reduce( &Local, &Global, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
                    root, MPI_COMM_WORLD )
   if (myRank == root) {
      globalBestNode = Global.bestNode
   }
   MPI_Bcast( &globalBestNode, 1, MPI_DOUBLE, root, MPI_COMM_WORLD )
}
```

Figure 14. MPI reduction computation for parallel IOR algorithm.

As the MPI and parcel code are structured similarly, this intermediate step to development was valuable to obtaining the correct parallel version.

4. Parallel Graph Clustering Algorithm Simulation

Performance measurement on the PIMs takes a significant amount of work; we anticipated that we would have limited time to evaluate alternative implementations of the Parallel InOutRatio Algorithm. To facilitate experimenting with alternative algorithms and algorithm parameters, we developed a MATLAB graph clustering framework to quickly prototype a variety of alternative parallel implementations.

We simulated the parallel implementation in PIMs to achieve the following goals:

1. Compare the result of the simulated version vs. actual parallelization for correctness.
2. Measure and compare a set of performance-related indices to compare different algorithms and communication strategies.

We have developed a prototype of the specific parallel algorithm from the previous section and minor variations of it, and instrumented this implementation to gather performance-related indices. We also used the MATLAB simulation environment to analyze the data sets from the ARDA EAGLE program, where ARDA and EAGLE are abbreviations for “Advanced Research and Development Activity” and “Evidence Assessment, Grouping, Linking, and Evaluation”, respectively.

Analysis of Sample Data Sets

The data sets evaluated for group (cluster) detection are derived from synthetic data developed by Information Extraction & Transport, Inc. These datasets were created by running a simulation of an artificial world whose main design focus was to produce datasets with large amounts of relationships between entities (as opposed to data with a large number of entity properties). The artificial world consists of *individuals* that belong to *groups* and exploit *targets*. Groups can be *threat groups* or *non-threat* groups that exploit targets in threat and non-threat. Individuals are *threat* individuals or *non-threat* individuals. Every threat individual belongs to at least one threat group. Non-threat individuals belong to non-threat groups. Threat groups have only threat individuals as members. Threat individuals can belong to non-threat groups as well.

Dataset	#of Nodes (People)	#of Links	Avg. Fanout	Phone Calls	Telecons	Telecon Respondents	Avg. Telecon Partipants	Avg. Telecon Links (complete graph)	Total Telecon Links
EASY_PROXY	1022	411026	804.4	133724	11720	74975	7.4	23.7	277302
Y3_DATASET_4018	2504	478047	381.8	153520	13479	87037	7.46	24.1	324527
Y3_DATASET_4019	10040	402151	80.1	147861	14184	78137	6.51	17.9	254290
Y3_DATASET_4020	12399	456040	73.6	152520	14113	85771	7.08	21.5	303520
Y3_DATASET_4021	3469	565977	326.3	162578	15128	103172	7.82	26.7	403399
Y3_DATASET_4022	98786	7591891	153.7	2952081	256648	1420244	6.53	18.1	4639810
Y3_DATASET_4023	101630	5423828	106.7	1826185	153916	978216	7.36	23.4	3597643
Y3_DATASET_4024	99405	3928352	79.0	1553019	137027	741217	6.41	17.3	2375333
Y3_DATASET_4025	9896	6534616	1320.7	2210888	208940	1243756	6.95	20.7	4323728
Y3_DATASET_4026	98715	4443255	90.0	1740857	102140	693681	7.79	26.5	2702398
Y3_DATASET_4027	987	7130817	14449.5	1909397	232232	1445500	7.22	22.5	5221420
Y3_DATASET_4028	9820	292955	59.7	122295	6145	42828	7.97	27.8	170660
Y3_DATASET_4029	9696	447616	92.3	125187	8407	69546	9.27	38.4	322429
Y3_DATASET_4030	19207	6093580	634.5	1954447	183996	1145593	7.23	22.5	4139133
Y3_DATASET_4031	16475	352556	42.8	121152	10362	64263	7.2	22.3	231404
Y3_DATASET_4032	15184	378151	49.8	110208	9571	66991	8	28.0	267943
Y3_DATASET_4033	10106	387739	76.7	129087	10968	70040	7.39	23.6	258652
Y3_DATASET_4034	1001	4662298	9315.3	1757329	107008	736795	7.89	27.1	2904969
Y3_DATASET_4035	1019	6159045	12088.4	1845146	126810	984503	8.76	34.0	4313899
Y3_DATASET_4036	9960	319464	64.1	116393	9903	58661	6.92	20.5	203071
Y3_DATASET_4037	10077	357262	70.9	128671	6306	50633	9.03	36.2	228591
Y3_DATASET_4038	999	7509741	15034.5	1929505	178686	1325649	8.42	31.2	5580236
Y3_DATASET_4039	1005	7746349	15415.6	2120496	190132	1370658	8.21	29.6	5625853
Y3_DATASET_4040	10047	321574	64.0	109490	9236	58143	7.3	23.0	212084
Y3_DATASET_4041	1011	6220428	12305.5	1856964	173629	1147197	7.61	25.1	4363464
Y3_DATASET_5046	13236	486321	73.5	160688	14639	90596	7.19	22.2	325633
Y3_DATASET_5047	2532	444121	350.8	138926	12670	81834	7.46	24.1	305195
Y3_DATASET_5048	13756	580285	84.4	191273	18000	109682	7.09	21.6	389012
Y3_DATASET_5049	988	401210	812.2	145145	7784	59366	8.63	32.9	256065
Y3_DATASET_5050	1008	5377252	10669.1	1881901	167128	1000560	6.99	20.9	3495351
Y3_DATASET_5051	9983	5840910	1170.2	1890410	178338	1101209	7.17	22.2	3950500
Y3_DATASET_5052	986	7579064	15373.4	1988612	180644	1333729	8.38	30.9	5590452
Y3_DATASET_5053	1002	8352377	16671.4	2138445	198368	1474070	8.43	31.3	6213932
Y3_DATASET_5054	990	4922863	9945.2	1815853	114619	788580	7.88	27.1	3107010
Y3_DATASET_5055	9967	7031532	1411.0	1810730	222562	1417211	7.37	23.5	5220802
Y3_DATASET_5056	1022	5793970	11338.5	1935004	131240	942947	8.18	29.4	3858966
Y3_DATASET_5057	1011	7691295	15215.2	2002565	197638	1403973	8.1	28.8	5688730
Y3_DATASET_5058	100301	3359485	67.0	1362705	117337	628379	6.36	17.0	1996780
Y3_DATASET_5059	99234	5860723	118.1	1969722	174317	1080802	7.2	22.3	3891001
Y3_DATASET_5060	99092	3335785	67.3	1483291	79730	505101	7.34	23.2	1852494
Y3_DATASET_5061	100507	7280506	144.9	2829247	245918	1361766	6.54	18.1	4451259
Y3_DATASET_5062	9897	348054	70.3	137533	11119	63088	6.67	18.9	210521
Y3_DATASET_5063	9998	488142	97.6	114363	12734	91408	8.18	29.4	373779
Y3_DATASET_5064	9939	408214	82.1	117006	9870	71044	8.2	29.5	291208
Y3_DATASET_5065	16046	308471	38.4	144141	7223	45245	7.26	22.8	164330
Y3_DATASET_5066	16743	367904	43.9	114073	10027	66509	7.63	25.3	253831
Y3_DATASET_5067	9970	399881	80.2	136443	10917	70579	7.47	24.1	263438
Y3_DATASET_5068	14209	435478	61.3	135433	11838	78573	7.64	25.3	300045
Y3_DATASET_5069	9907	390022	78.7	150192	7683	56986	8.42	31.2	239830
min	986	292955	38.4	109490	6145	42828	6.36	17.0	164330
avg	22587	3185686	3413.2	1023729	88429.2	572784.55	7.5818	25.2	2161958
max	101630	8352377	16671.4	2952081	256648	1474070	9.27	38.4	6213932

Figure 15: Summary of ARDA/Eagle Data Sets.

Figure 15 above illustrates data characteristics. Most of the simulations use Y3-DATASET_4028, selected in the table. We see from the table that there is a huge discrepancy among the datasets in terms of connectivity and size. This point is illustrated in the next three figures. In Figure 16, each axis represents one dimension in the dataset. However to make these graphs comparable to each other we normalized the axes over all datasets. Hence a dataset with maximum number of nodes gets 1 in *#of Nodes* axis and the dataset with highest number of links gets 1 in *#of Links* axes.

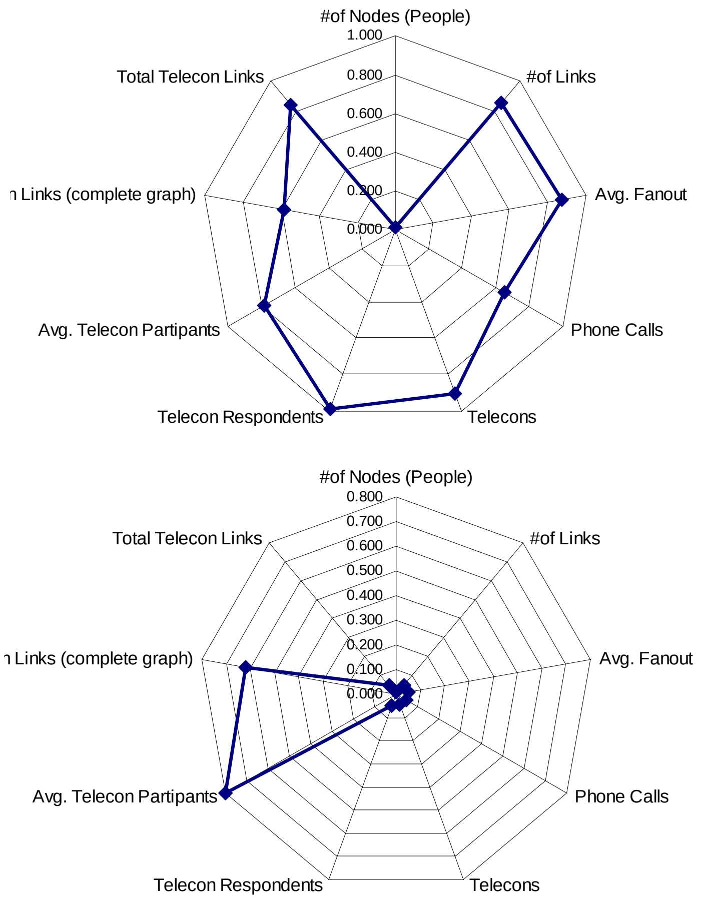


Figure 16: Representation of 2 data sets, demonstrating their differences.

Figure 17 shows a scatterplot of all the datasets, comparing number of links to number of nodes. It is shown on a log-log scale to highlight behaviors when either number of links or number of nodes is small. As you can see from this data, there are many variations in dataset properties – some have large numbers of nodes but small numbers of links, others have lots of links and fewer nodes, and a few have both large numbers of nodes and large numbers of links. This sample suggests that the best parallel algorithm for one data set may not be the best for all, a topic of future work.

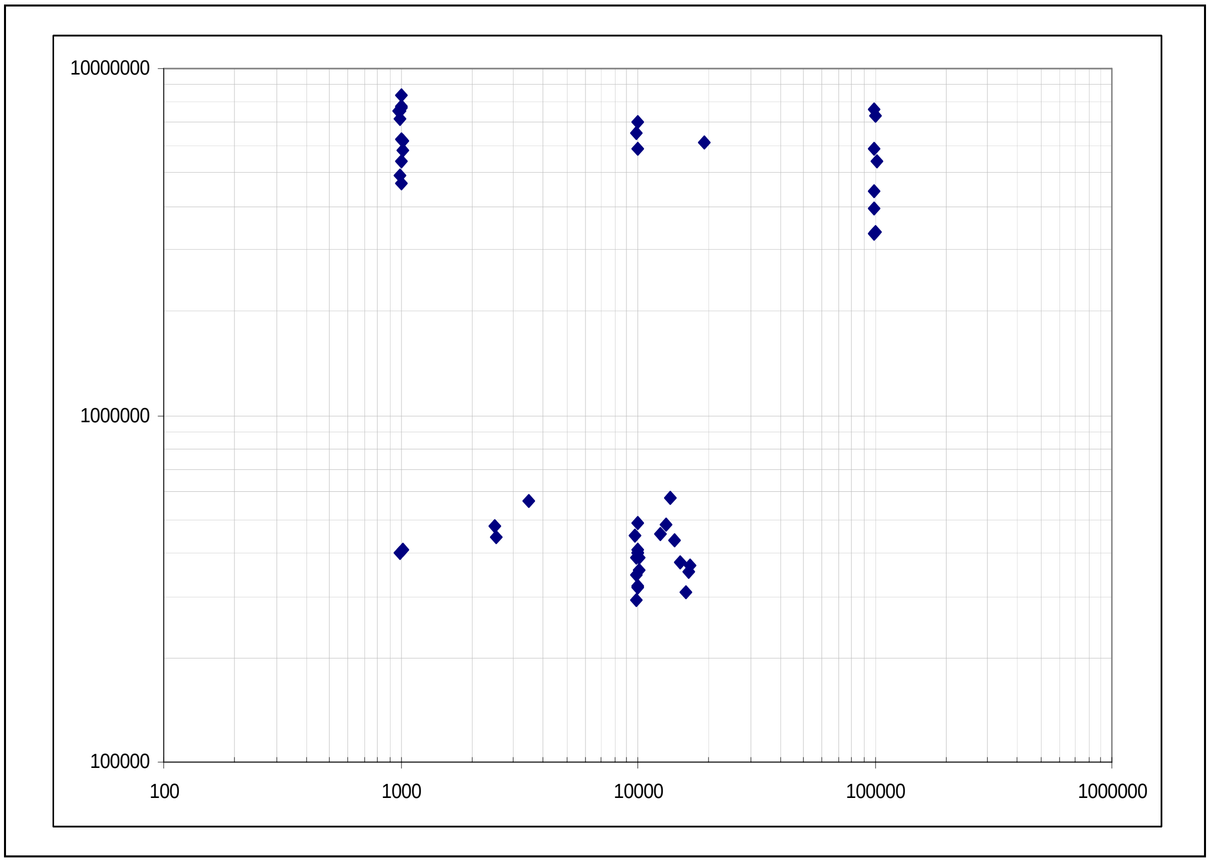


Figure 17: Scatterplot of all data sets.

To run the IOR over a set of PIMs we split the Link Table and distribute the load over all PIMs, as described in the previous section. In addition each PIM has a copy of the set of nodes in the initial cluster (*seed members*) and the links among them (*seed links*). At each iteration each PIM finds the node with the highest IOR and reports the result to other PIMs. Using this basic approach, we identify a couple of variations on the algorithm in Section 4. First, we can evaluate the impact of reporting one or more members at each iteration. (We call this parameter **M**.) Obviously, if the number of members reported by each PIM is more than one, the overall accuracy of IOR decreases, but parallelism increases and communication costs are reduced. As with other such algorithms, small differences in results can be tolerated. We can also vary the group size, a parameter we call **N**. We can also vary the number of PIMs. We assume PIMs communicate with each other hierarchically. Each PIM is located in a leaf of a balanced tree and reports the best node to its parents. The results of these variations are shown in this section.

Experimental Setup

We ran our simulation of IOR on a group of 1, 2, 4, 8 and 16 PIMs over the Y3-DATASET_4028 of Figure 15. The following are the main parameters for this experiment.

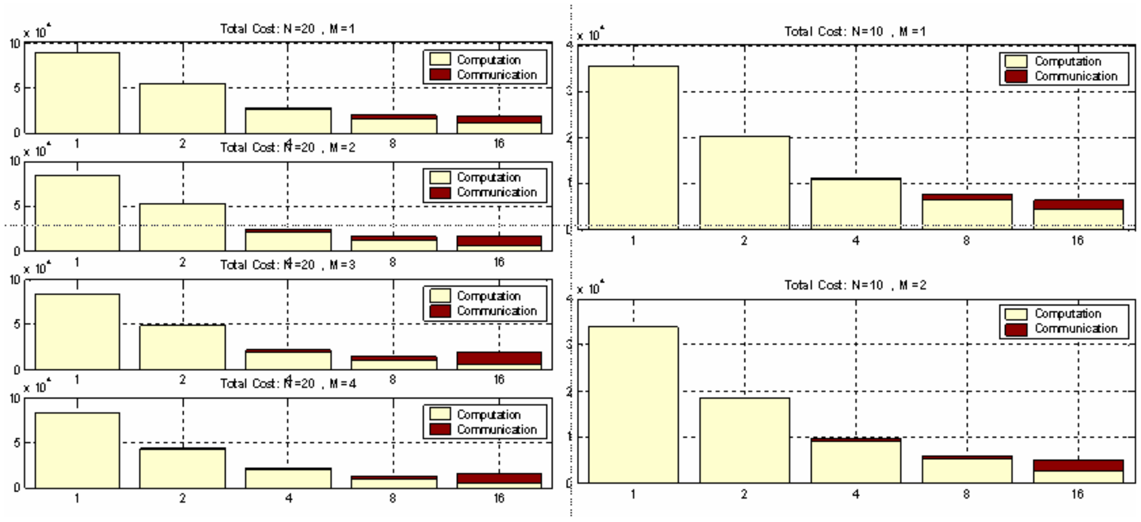


Figure 18: Processing Cost vs. Communication Cost. For 1,2,4,8 and 16 PIMS. Left: N = 20, M = 1,2,3,4. Right: N = 10 and M = 1, 2.

Parameter	Definition
A	Number of PIMs in the simulation {1,2,4,8,16}
N	Number of new cluster members to be found {10,20}
M	Number of new cluster members to be reported by each PIM at each round ={1,2}

Cost

There are two type of costs associated with PIM execution, *processing* and *communication*. We represent *processing cost* with the total number of links accessed by all PIMs. Because we are prototyping the algorithm in MATLAB, this measurement gives an indication of how the amount of work is affected by different algorithm strategies. Any direct execution time performance measurements of the MATLAB prototype would not necessarily be representative of a production parallel implementation. We also represent the *communication cost* with the total number of links passed from one PIM to another.

Processing cost

In each step In/Out ratio treats a node as a potential new cluster member. The *processing cost* is calculated through the following steps:

1. Count number of all links connected to this node
2. Count “in” links between the new node and nodes in current seed cluster (In List)
3. “out” links are the differences between 1 and 2
    - 1 → can be calculated once in O(F) time (F = avg. fan out)
    - 2 → calculated in each iteration in O(F) time

Communication Cost

Communication cost represents the aggregate message passing among PIMS when the best In/Out ratio is calculated (reporting up) and when the best ratio is reported back to all PIMs (reporting down).

- Reporting Up
  - The best node from each PIM plus relevant rows from the link table
  - Cost: the number of rows associated with the best local node
- Reporting Down
  - The selected node and relevant rows from link table are sent to all PIMs
  - Cost: the number of rows associated with the best global node

Figure 18 compares the cost (processing and communication) across all architectures for group size equal to 10 and 20. As is expected, the communication cost increases when the number of PIMs increases.

Load Balance

In this section we briefly address the following issues regarding load imbalance.

- Why there is load imbalance and how big is it
- Sensitivity to group size and number or members reported at each round
- Is there an opportunity to overlap computation and communication (to reduce the communication cost)?

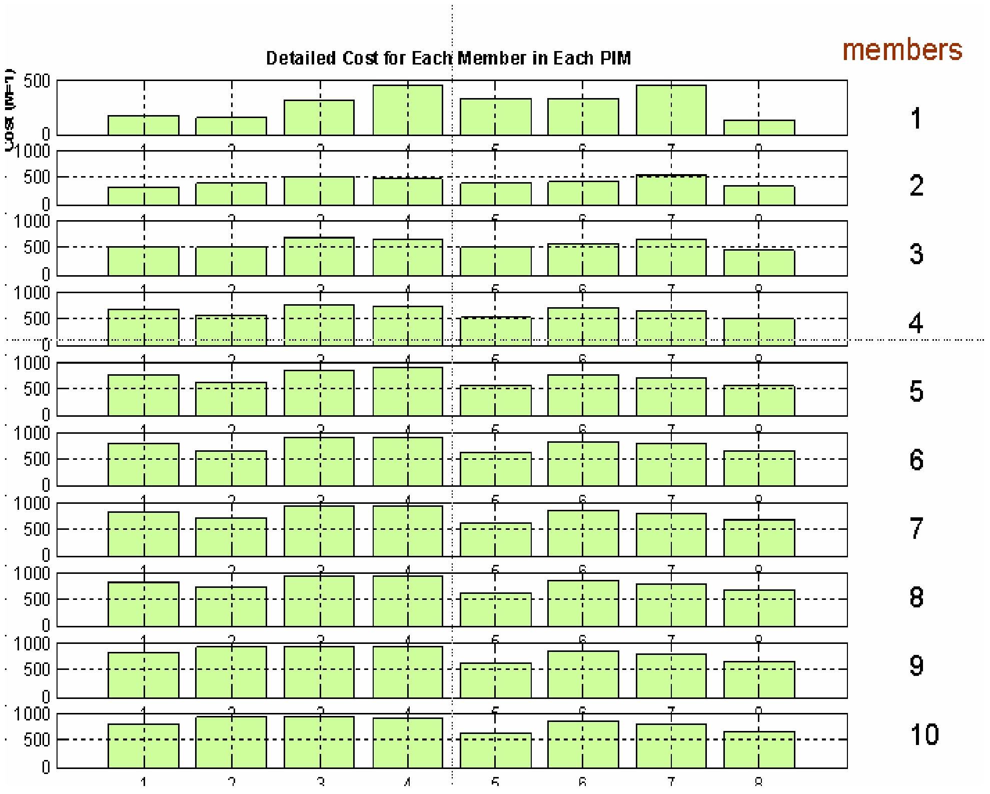


Figure 19: Load imbalance in 8 PIMs, Group Size = 10, M = 1 (communicate one)

Figure 19 and Figure 20 illustrate the load balance among 8 PIMs. The total number of new cluster members to be discovered is 10. M, the number of newly discovered members per step is set to one in the upper graph, and two in the lower graph. Each row in the figure shows a step of the ten-step computation, and work on PIMs is represented horizontally.

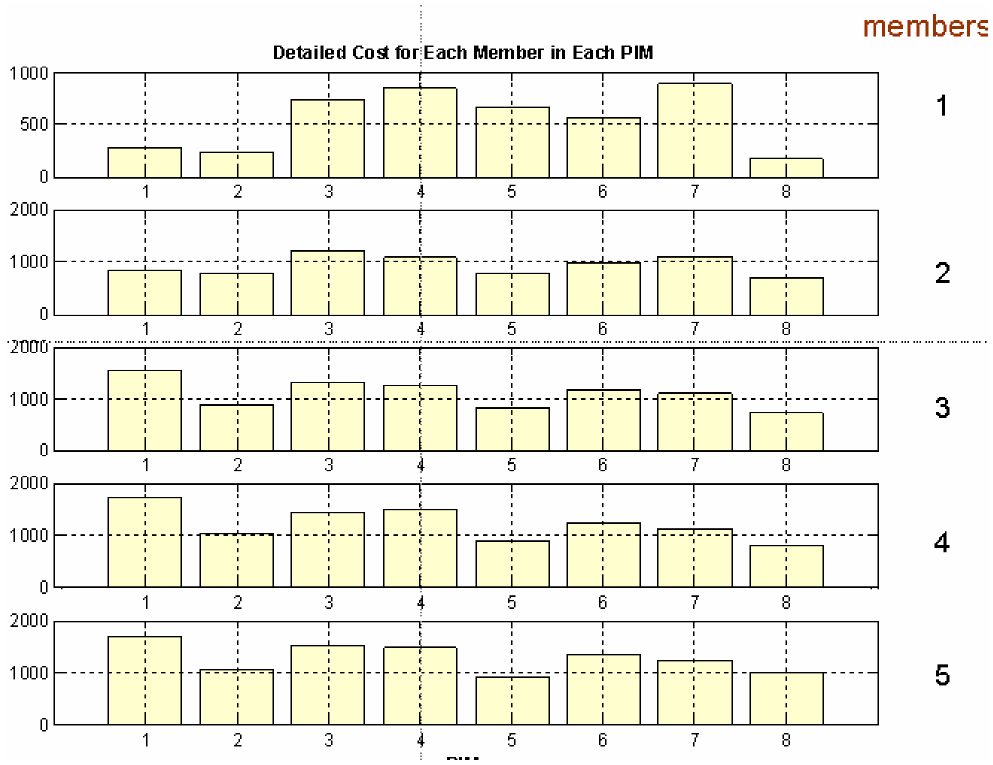


Figure 20: Load imbalance: 8 PIMs, Group Size = 10, M = 2 (communicate two)

From Figures 19 and 20, we see that the amount of work at each PIM varies significantly, and also varies across different time steps. However, the trend is toward similar behavior at each time step, related to connectivity of the subgraph allocated to a PIM. Thus, there is load imbalance, suggesting that various strategies for managing load might be useful for this algorithm, such as virtualization and dynamic scheduling. In addition, with increase in the number of seed member's each PIM will be loaded to somewhat similar link table (especially in a scale free network there are a couple of nodes with extremely high degree) and load imbalance reduces consequently. Another important observation is that when the amount of load imbalance increases there is enough time for some of the PIMs to communicate with each other.

Table 2 summarizes lessons learned from this experiment.

1	Seed Member Size	↑	Load Imbalance	↓
2	M (members reported at each round)	↑	Load Imbalance	↑
3	Uniformity in the data	↑	Load Imbalance	↓
4	Load Imbalance	↑	Communication Cost	↓
5	Load Imbalance	↓	Computation Cost	↓

Table 2: Summary of Load Balance Experiment

Table 3 summarizes the cost of computation and communication (as the number of accessed links) for increasing numbers of PIMs, following a hierarchical policy for communication. Comparing Figure 18 and Table 3 shows in general increasing the number of PIMs reduces the cost of overall computation. However, the benefits depend on the actual cost of communication relative to computation cost.

PIM		**1**	**2**	**4**	**8**	**16**
Computation Cost	M=1	35373	20209	10866	6675	4602
	M=2	33923	18427	9251	5078	2770
Communication Cost	M=1	0	123	369	861	1845
	M=2	0	138	369	861	2250

Table 3: Computation and Communication cost for 1, 2, 4, 8, 16 PIMs for M = 1, 2.

Connectivity Effect

Connectivity is an important feature of a graph which refers to the number of links in the graph. In this experiment we measured the effect of connectivity on the processing cost. As is illustrated in Figure 21, the processing cost increases when the connectivity increases. However this rise in cost is smaller for 16 PIMs than for smaller numbers of PIMs. Figure 22 illustrates the effect of connectivity on speedup. As it shows, 16 PIMs perform even better as the connectivity increases. For high connectivity performance improvements trail off due to load imbalance among PIMs for this data set.

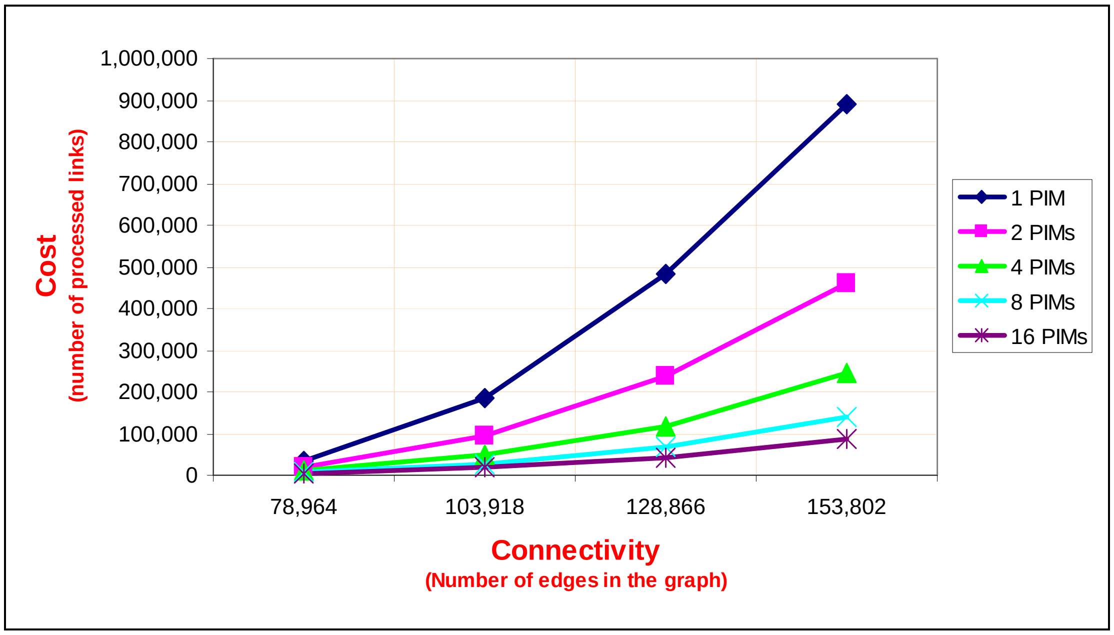


Figure 21: Cost vs. Connectivity

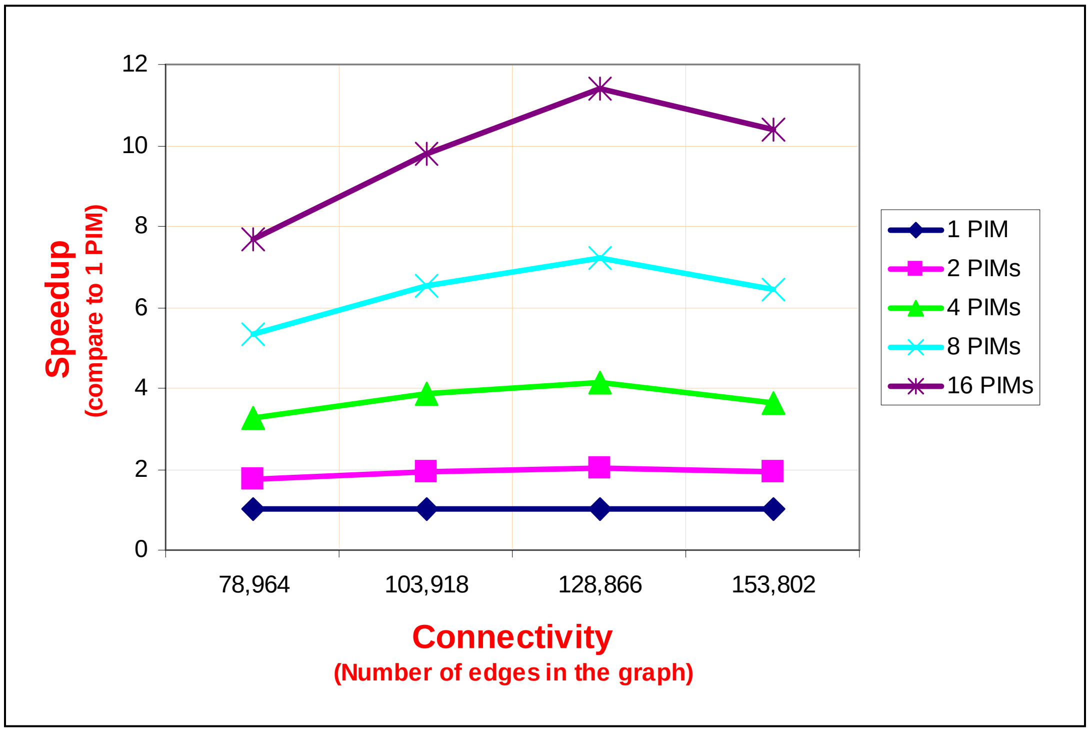


Figure 22: Speedup vs. Connectivity

Compromising Accuracy for Speedup

To minimize the number of communications we may report more than one potential new member at each round. Table 4 illustrates the change in accuracy of the result when M (the number of members reported in each round by each PIM) changes from 1 to 2.

For actual evaluation one should compare the result of in/out ratio with ground truth (actual members of the cluster). However, since access to ground truth is not possible in many real world datasets we only measure the deviation of the final result from M = 1 (when we report only one member at each time)

PIM	1	2	4	8	16
N = 10 **M = 2**	1	.75	.75	.64	.25
N = 20 **M = 2**	1	.41	.36	.36	.16

Table 4: Result Deterioration (F-value compared to the M =1 case)
N: number of desired members to be discovered
M: number of members reported in each round by each PIM

The result of Table 4 is that the deterioration of the result grows with M and N and number of PIMs. Only for small numbers of PIMs with N=10 are the results within an acceptable range of accuracy.

5. Performance Analysis on PIMs

Figure 23 below illustrates our approach to performance evaluation of codes on the PIM hardware testbed. The process has many steps. Initially, scalar code for a single PIM is developed that implements a specific algorithm. As an initial test, we compile and assemble the code, without any optimization (*i.e.*, gcc –O0), and validate its execution on the hardware, using very small TEST INPUT data. To collect performance measurements, the process is far more complex. The code must be highly tuned. Also, the data sets should be as representative of realistic data sets as possible within the constraints of the system (referred to as SCALE INPUT). It is important that the SCALE INPUT be large enough to stress the Itanium-2 memory system, since this is the regime where PIMs will be advantageous. To optimize for single PIM performance, we use the frontend compiler described in Section 2 that converts sequential code to SIMD code to exploit the hardware's WideWord capability. We also use the optimization within the gcc backend (*i.e.,* gcc –O2) to further optimize the code, performing register allocation, constant propagation, strength reduction, and other standard optimizations.

Our performance measurements are compared against the Itanium-2. For this experiment, the host-only versions were compiled with the icc 8.0 C compiler with -O3 options available from Intel, running under Linux version 2.4 on the single processor 900MHz HP zx6000 workstation.

To develop parallel code, we add explicit parallel constructs to the code to communicate between PIMs, and optimize the data/computation partitioning by hand.

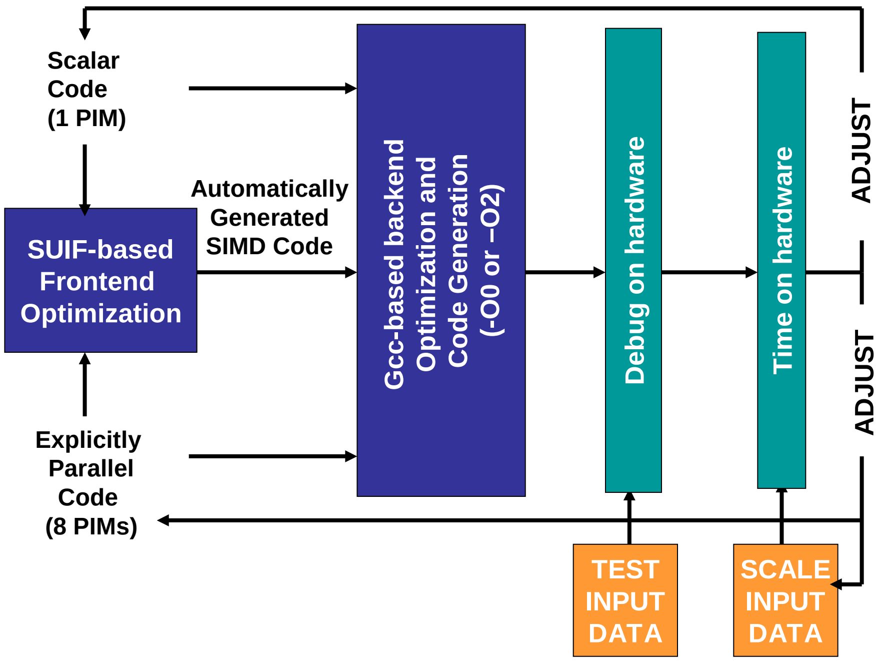

Scalar Code (1 PIM)
SUIF-based Frontend Optimization
Automatically Generated SIMD Code
Explicitly Parallel Code (8 PIMs)
Gcc-based backend Optimization and Code Generation (-O0 or –O2)
Debug on hardware
Time on hardware
ADJUST
ADJUST
TEST INPUT DATA
SCALE INPUT DATA

Figure 23. Performance Tuning Process on Hardware.

Due to the complexity of the code to be analyzed, it was our goal to minimize any manual intervention in the generation of code. While this goal was challenging given the shot timeline of this project and its limited scope, we believe it is important to the credibility of the resulting measurements. Thus, we have invested in improving the compiler implementations rather than handcoding in assembly. Only very minor handcoding was performed on compiler-generated code – to enhance instruction scheduling in StreamAdd and to optimize control flow in MI.

Bandwidth Benchmarks

StreamAdd

We present performance measurements for the implementations of StreamAdd and Random Access described in Section 3. Performance results for StreamAdd, a subset of STREAM from the HPC benchmarks, are shown in Figure 24. The y-axis represents execution time, and the x-axis shows performance as a function of data set size. There are four curves. The one labeled Itanium2-icc8 represents the code executing on the Itanium-2 host. The one labeled PIM.compiled shows execution time of the same data set on 1 PIM, and is generated by the compiler. The one labeled PIM.tuned represents the hand-tuned version of the code. The fourth curve projects execution time on 8 PIMs. As there is no communication in this code, the PIM performance speeds up linearly with the number of PIMs.

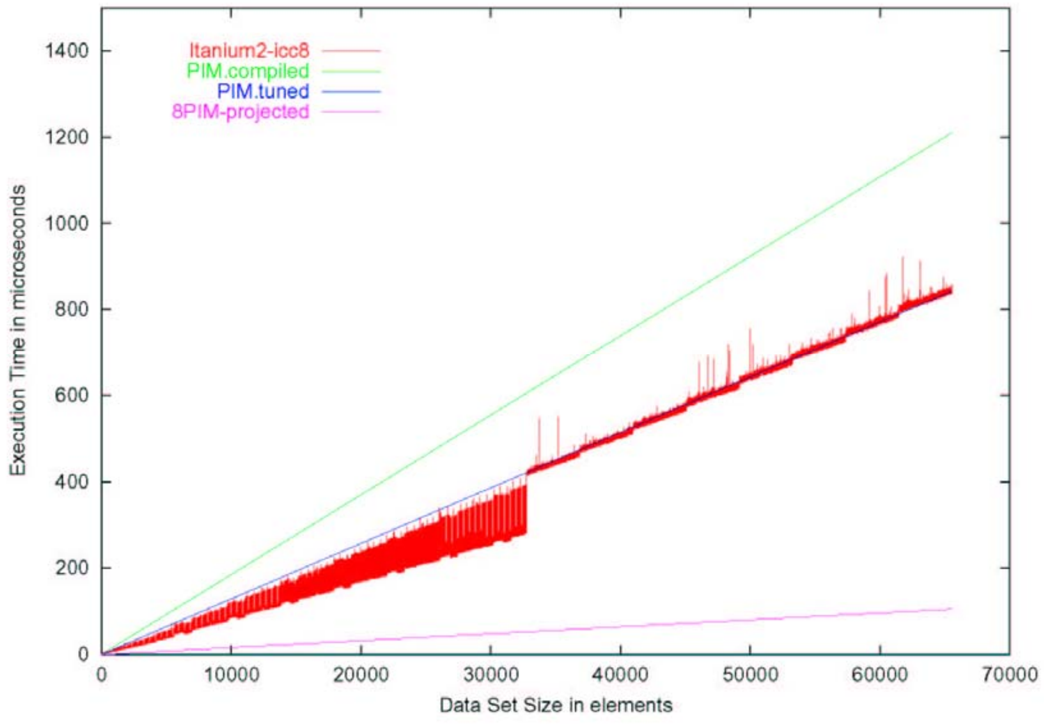


Figure 24. StreamAdd Performance Measurements.

The PIM floating-point performance for StreamAdd shows deterministic performance monotonically increasing as the problem size increases. The Itanium2 result shows performance that varies as a function of problem size. It is better for smaller problem sizes, but as the problem sizes get larger, the way in which the system allocates memory leads to worse performance.[1]

We observe in Figure 24 that the single PIM execution time (pim) is comparable to that of the Itanium2 execution time (zx6000). With eight PIMs running in the memory system, there is an 8x speedup over the single Itanium 2 processor (8pim-projected).

Random Access

We mapped the RandomAccess code to the hardware implementation using the approach outlined in Section 3. Because this application requires host and PIM collaboration at a fairly fine grain, it was substantially more difficult than StreamAdd to achieve a working implementation, and stressed the host-to-PIM interface. The process

[1] Although the details of this performance result are beyond the scope of this report, we pinpointed this behavior using the Itanium2 hardware performance monitoring support and discovered it has to do with how data is laid out in memory and resulting cache behavior.

of writing from host to PIMs is complex. To send a 256-bit parcel to a PIM requires reordering bits and a stream of writes due to the spreading of data across multiple memory chips and bit interleaving by the Itanium-2 memory controller. We write a subset of the bits on each memory operation, and must rearrange the bits on arrival at the PIMs. While this works in our system, we envision a system where this sort of reorganization is not required, and we do not include the cost of this reorganization in our timings.

Instead of including end-to-end performance, we show performance of the host and PIM portions of the implementation, as measured on the hardware, and combine these measurements. On the PIM side, the code was instrumented and measured from the time a parcel is sensed through the update until the time it is waiting again for another parcel. Measurements are made in terms of clock cycles. The performance of the PIM across timing measurements has proven to be deterministic and consistent. On the host side, we replaced the writes to the host parcel buffer on the PIMs with a conventional write. To force the write to occur, we flush the cache line in which the PIM payload resides just prior to the point where the parcel should be sent. We verified with a logic analyzer that this implementation had the appropriate number of memory operations.

Performance measurements are shown in Figure 25. Using the code of Section 3, with four <*ran*, *offset*> pairs sent per parcel, the host will be able to send parcels at the rate of 2.5M parcels per second (we discuss below how to increase this rate). A single PIM can process parcels at the rate of 1.2M parcels per second. In a system with two PIMs, the PIMs and host can process parcels at the same rate. Beyond this, the rate at which PIMs process parcels is much faster than the host's ability to produce parcels, and the host system bus becomes a performance bottleneck.
.

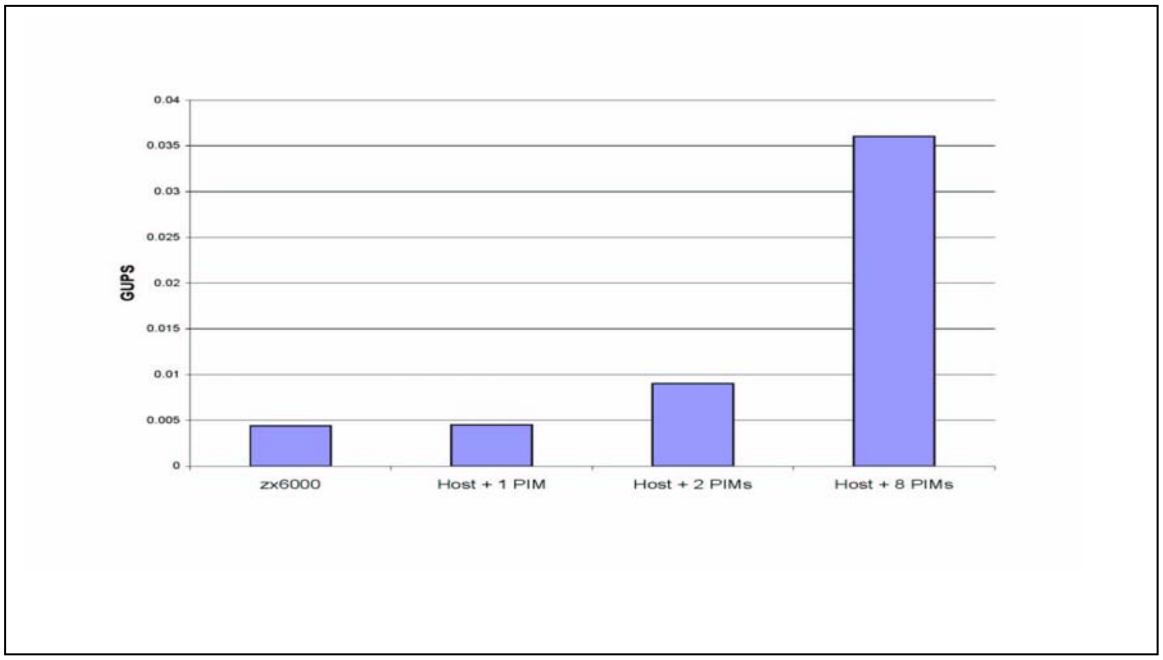


Figure 25. G-updates/second for RandomAccess.

Using these results, we can project end-to-end performance as shown in Figure 25. The first bar is the measurement of the host-only implementation on the zx6000. The performance is 0.0044 GIGA Updates Per Second (GUPS.) If the Host+PIM code is used with just one PIM, the performance is comparable, 0.0045 GUPS. With a second PIM in the system, we can achieve a 2X speedup, achieving 0.01 GUPS.

The 8 PIM results rely on sending parcels to 8 PIMs in parallel. This is possible by exploiting what is in some sense a challenging aspect of the host-to-memory interface. A write to a particular PIM must be done 4 bytes at a time, with the remainder of the bytes ignored by the memory system. If the address space is arranged appropriately, the remainder of the bits can go to distinct PIMs, four bytes at a time. This would require a slightly different implementation, such that when the buckets are full for one PIM, an aggregate communication would be performed for all PIMs. This complex arrangement of bytes/addresses could be transparent to the user by implementing it in the boot and communication libraries. With such an implementation, we could achieve 0.038 GUPS.

Scaling the PIM clock rate up by 2x is easy and will then provide a 2x speedup on GUPs with just one PIM since it will then match the single PIM memory lane parcel production rate. That is true for each additional PIM also. But increasing PIM speed beyond that does no further good since the bandwidth per 32 bit PIM lane is fixed by the Host production (frequency of parcel buffer writes) number of 2.54 Mparcels/sec. The zx6000 cannot produce more parcels through that 4 byte wide memory lane, but there are 7 other lanes that can be used when all host parcel buffers are written in parallel. At that point a 16x speedup can be reached. With double speed PIMs filling the entire memory bus width, PIMs can consume the maximum parcel bandwidth the zx6000 can produce.

Mutual Information on a Single PIM

	Execution Time	**Cycles**	**Instructions Per Clock**
Itanium-2	5.5ms	4.9M	1.588
Single PIM (scalar)	113.0ms	16M	<1
Single PIM (superword, compiler-generated)	94.6ms	13M	<1, but includes SIMD ops
Single PIM (superword, hand tuned)	32.1ms	4M	<1, but includes SIMD ops

Table 5. Performance comparison for MutualInformation on TEST input.

We present numbers on the TEST data set of 520 nodes and 4768 links, which was derived by deleting nodes and links from a larger data set, and its purpose was to derive a test input small enough to be conveniently loaded onto a single PIM. It is roughly 500Kbytes. Performance results for the Itanium-2 and for a single PIM on the TEST input are shown in Table 5.

The first column identifies whether the code was run on the Itanium-2 or the PIMs. The execution time is shown in the next column. Recall that the Itanium-2 clock is operating at 900 MHz while the PIM clock rate is 140MHz, and the Itanium-2 is 6-way Very Long Instruction Word (VLIW) while the PIM processor is a single-issue, in–order processor. Thus, to facilitate comparison between the two systems, the number of cycles is shown in the next column, followed by instruction-per-clock in the fourth column.

There are three versions of code for the PIMs in Table 5. The first uses the scalar processor, and does not exploit the PIM's WideWord unit. The second is automatically generated from sequential code by the SUIF compiler frontend to perform SIMD operations in the WideWord unit. The third was a result of hand tuning the second version. There is a modest speedup of 1.2X in the Wide version of the code over scalar code. However, with minor tuning having to do with control flow optimizations as discussed in Section 2, there is a speedup of 3.5X over the scalar version of the code.

In comparing the Itanium-2 performance to the PIMs, the PIM code runs slower, but it is executing 18% fewer cycles than the Itanium-2. The performance difference is largely due to the Itanium-2's 6X faster clock rate. In addition, the Itanium-2 memory system is performing well for this relatively small data set. It is essentially operating completely out of its L2 cache.[2] (Note that this is somewhat an artifact of the fact that the Itanium-2 does not allocate floating point data, such as the conditional probability results, to L1 cache.) Even though the L1 miss rate is somewhat high, there is plenty of work to keep the processor busy, achieving an Instructions-Per-Clock of 1.588. This means that the Itanium-2 is retiring more than one instruction per cycle, while the single-issue, in-order PIM processor is retiring less than one instruction per cycle.

The TEST input was very small, so we were interested in how the MutualInformation kernel would perform on the larger data sets of Section 3, Figure 15. We selected three datasets from Figure 15, as shown in Table 6 below. The first column describes the data set, with the TEST input included for reference. The second column provides how many nodes and links are in the graph associated with each dataset, followed by its size in bytes in the third column. Itanium-2 execution time for each data set is in the fourth column, and Itanium-2 IPC is shown in the fifth column. The sixth and seventh columns show L1 and L2 miss rates, respectively.

[2] Because the code in Section 3, Figure 9 was designed to expose fine-grain and coarse-grain parallelism, the sequential Itanium code is slightly different in its loop structure, and the size of the intermediate structures *pxv, aux1, aux2, and cpyxt.* Shown in Figure 9, they can be quite large, but the original implementation used on the Itanium2 keeps them small, the size of O(ntypes^2), and performs the complete MI computation for each link.

Data Set	Nodes and Links	Size (in bytes)	Execution Time	Instructions Per Clock	L1 Miss Rate	L2 Miss Rate
TEST	520 nodes 4764 links	499KB	5.5ms	1.588	20.37%	0.51%
Y3_4027	987 nodes 3.368M links	350MB	1.719 s	1.478	21.50%	0.66%
Y3_4028	9820 nodes 387668 links	40MB	332.4ms	1.540	22.52%	0.57%
Y3_4036	9960 nodes 384240 links	40MB	408.2ms	1.561	31.84%	0.48%

Table 6. Mutual Information Data Scaling on Itanium-2

Overall, the table reveals that this kernel behaves similarly regardless of execution time or data set size. While there is some variation in IPC across the different data sets, it is generally high, ranging from 1.478 to 1.588. L2 miss rates are extremely low, less than one percent. L1 miss rates are somewhat high, but the L2 accesses comprise over 80% of memory references across all the different data set sizes.

In summary, this kernel performs well on the Itanium-2's memory system, and there is little room for improvement. The reduction in PIM cycles as compared to the Itanium-2 cycles is apparently a result of optimizations for the PIM's WideWord unit.

Graph Clustering, Sequential and Parallel

	Execution Time	Cycles	Instructions Per Clock
Itanium-2	0.26ms	233K	0.806
Single PIM (scalar)	1.11ms	155K	<1
2 PIM parallel	1.65ms	231K	<1

Table 7. Performance comparison for Graph Clustering on TEST input.

For graph clustering, we present numbers on a TEST data set of 9820 nodes, 23656 links, and 9704 pairs. Note that this data is organized more compactly than for MutualInformation. It is represented by pairs of nodes, rather than by links. As in the previous example, this data set was derived by deleting nodes and links from a larger data set, and its purpose was to derive a test input small enough to be conveniently loaded onto a single PIM. It is roughly 273Kbytes. Performance results for the Itanium on the Itanium-2, for a single PIM running sequential code and 2 PIMs executing the parallel code on the TEST input are shown in Table 7 above. A version of the code with SIMD instructions for the Wideword unit was generated by the compiler and validated in simulation, but we did not have time to debug it on the hardware.

The columns are as in Table 5. For this code, the IPC on the Itanium-2 is much lower at 0.806. As a result, the scalar single PIM code (without any WideWord code) is only 4.26X slower than on the Itanium-2, and executes a third fewer cycles than the Itanium-2. This improvement in cycle count is an indication that the memory behavior of this code on the Itanium-2 is limiting its performance.

We developed the parallel version of the algorithm as described in Section 3 and show preliminary performance results for two PIMs. The performance is slower than the scalar code, and while we did not have time to investigate the reason, we suspect it is the usual problem that the overhead of the parallelization is too high for this data set size and only 2 processors. Usually speedups can be improved by scaling up the data set size, but we are limited by the 1MByte storage on each PIM.

To investigate how data scaling impacts performance of the graph clustering benchmark, we performed measurements on the same three data sets as in Table 6, shown in Table 8 below. The data has been translated to use the compact pair representation.

Data Set	**Nodes and Links**	**Size (in bytes)**	**Execution Time**	**Instructions Per Clock**	**L1 Miss Rate**	**L2 Miss Rate**
TEST	**9820 nodes 23656 links 9704 pairs**	273KB	2.59ms	0.806	19.6%	14.2%
Y3_4027	**987 nodes 3.368M links 769298 pairs**	15MB	7.234s	0.489	15.7%	39.9%
Y3_4028	**9820 nodes 387668 links 157928 pairs**	3MB	219.7ms	0.653	31.5%	26.5%
Y3_4036	**9960 nodes 384240 links 196544 pairs**	4MB	592.5ms	0.464	31.8%	25.1%

Table 8. Graph Clustering Data Scaling on Itanium-2

We see from Table 6 that performance of graph clustering is greatly affected by data set size. The data sets range from the 273KB of the TEST input, to 3 and 4MB for the intermediate sizes and 15MB for the large size. IPC is as low as 0.464. One interesting observation from this table is the difference in performance behavior of the Y3_4028 and Y3_4036 data sets. While they have a comparable number of nodes and links, the number of pairs is about 24% larger in the latter case, resulting in a 33% increase in data set size. This difference leads to a more than doubling of execution time, and a much lower IPC. While not shown here, the data set in Y3_4036 has a significant increase in L3 misses which appears to be the cause of the performance difference.

Our results on graph clustering are more preliminary than the others in this report, but these results demonstrate a potential for PIMs in graph clustering algorithms. Our scalar PIM code is not at all tuned, and yet there are fewer cycles than on the Itanium-2. A scalable parallel algorithm for graph clustering, which is an open research problem, is required to achieve speedups for large graphs.

Scaling Analysis and Projected Performance

The previously described measurements of these kernels on the prototype hardware are very valuable, but as it is an academic hardware implementation that was developed on a short time schedule, it does not achieve performance levels that we could expect from a commercial-quality PIM implementation. Table 9 below summarizes the differences between the Itanium-2 and PIM processors.

	Clock	**CPU Info**	**Area**	**Transistor**	**Power**
Itanium-2	900 MHz	EPIC, 6-way	421mm^2	221M	~100W
1 PIM	140 MHz	single-issue, in-order, pipelined	121mm^2	56.6M (55M memory)	~1W
8 PIMs	140 MHz	single-issue, in-order, pipelined	8 x 121mm^2	453M (440M memory)	~8W

Table 9. Comparison of Itanium-2 and PIM processors.

The key differences are that the Itanium-2 has a 6x faster clock rate, and it is a 6-way issue EPIC processor rather than the single-issue, in-order processor of the PIM chip. A further constraint on our PIM implementation is that the on-chip memory is just 1 Mbyte. On the other hand, the Itanium-2 consumes a maximum of 130 watts while each PIM runs at about one watt, that is, a single PIM has an over 100x power consumption advantage. In terms of performance/watt, our PIM produces 1.12 Gflops/watt while the Itanium-2 produces 0.036 Gflops/watt giving the PIM a 30x advantage in peak performance per watt. For these reasons, in this section we evaluate the measurements of the previous section in a broader context, to account for the constraints of an academic project, consider architectural variations and look to the impact of future technology trends.

The technology trend is toward denser memories and multi-core processors, not higher clock speeds or more complex processors. Thus we can expect that production PIM chip processor speeds can be on par with IBM Cell (which is multi GHz in 90nm). In the end, the best performance/watt may be achieved by slower clock speeds, and in fact, it may be best to have PIM processors operate no faster than the embedded DRAM frequency. These trends are reflected in Table 10, which was derived from the 2006 ITRS Road Map.

Metric	Unit	Year	
		2007	2010
Technology node	nm	65	45
Clock*	GHz	0.5	1
Processor/eDRAM cores	Per chip	16	32
Aggregate memory bandwidth	GB/s/chip	256	1K
Memory size	MB/chip	128-256	256-512

Table 10. Projections of commercial implementations of PIMs for 2007 and 2010.

Based on these projections, we scale the performance measurements from the previous section, and the result of this exercise is shown in Table 11 below. In the first column, we show the raw 1-PIM speedup over Itanium-2, which is in both cases less than one. The next column shows the impact of clock scaling, based on the following formula:

$$\text{Clock Scaling} = \frac{\text{IT2 Cycles}}{\text{PIM Cycles}} \tag{1}$$

That is, we assume the PIM and Itanium-2 have the same clock rate. In the next column, we also try to account for the impact of scaling to larger and more representative data set sizes. We normalize the speedup by adjusting the IPC from the TEST input set up to the SCALE input set, using the following formula:

$$\text{Data Scaling} = \frac{\text{IT2 Cycles} * (\text{IPC}_{\text{test}} / \text{IPC}_{\text{scale}})}{\text{PIM Cycles}} \tag{2}$$

In examining the results, we see that, since the graph clustering code started out closer to the Itanium-2 performance, it yields a 1.5X speedup in terms of reduction in cycles, as compared to 1.225X for Mutual Information. Further, since the IPC for the graph clustering code varies significantly with larger data sets, it shows an additional gain

	Raw 1-PIM Speedup	**Clock Scaling (1) above**	**Data Scaling (2) above**	**2 PIMs**	**Projected 8 PIMs**
MutualInformation	**0.171X**	**1.225X**	**1.316X**	**~2.632X (projected)**	**~10.528X**
Graph Clustering	**0.234X**	**1.503X**	**2.611X**	**1.752X**	**unknown**

Table 11. Projected performance accounting for clock and data scaling.

up to a 2.6X speedup as compared to a minor improvement for Mutual Information. We also considered parallel performance. The Mutual Information kernel is embarrassingly parallel as written, performing for the most part independent computation, and thus we project linear speedups for 2 and 8 PIMs. The 2 PIM results, even after clock and data scaling, does not perform as well as the scalar version, as discussed in the previous section. Given the complexity of the parallel implementation, we did not attempt to project 8 PIM performance for this kernel.

6. Summary, Conclusions and Future Directions

This project offered the first opportunity to derive performance measurements on the Godiva hardware testbed, a PIM-based architecture implementation developed at USC/ISI over a 7 year period. An important result of this project was that we were able to demonstrate the effectiveness of PIMs at delivering memory bandwidth on the bandwidth benchmarks StreamAdd and RandomAccess. For both kernels, our PIMs were able to deliver comparable bandwidth to the Itanium-2 with just a single PIM, and 8X more bandwidth with 8 PIMs, in spite of the Itanium-2's far more powerful and 6X faster processor. Further, RandomAccess demonstrated the advantage of supporting sharing of data between host and PIM, a unique feature of our PIM architecture as compared to others. While not included in our scaling results, we expect these kernels would be at least 6 times faster on a single commercial PIM than a conventional platform.

The link discovery kernels, which were the focus of this project, were much larger than the bandwidth benchmarks, and proved more difficult to demonstrate bandwidth gains. The results in Table 10 of the previous section suggest that, while there is up to a 10X improvement shown for the MutualInformation kernel using a commercial implementation of PIMs, it is largely due to parallelization and optimizations targeting the SIMD WideWord unit, which could be obtained on other architectures. The algorithm has data locality and operates almost completely out of the L2 cache, limiting the performance gain of PIM's nearby and conceptually larger DRAM. This could be, to some extent, the result of turning this computation into a kernel and abstracting away the gathering of data into the link table data structures. Constructing the link table could be the bigger limitation to memory system performance.

The results for the graph clustering kernel are far more promising for PIMs. As the graph gets larger, the kernel exhibits poor memory-system behavior on the Itanium-2. As a result, the number of cycles on the PIM is a third lower than on the Itanium-2 even with our 273KB TEST input. Our analysis suggests that much larger gains would be obtained for memory sizes in a commercial PIM implementation. The real open question is how to perform graph clustering in parallel and manage the communication overhead. Our parallel PIM results are preliminary, and did not show performance improvement for the small TEST data set on 2 PIMs. We augmented these parallel results with the simulations described in Section 4. We found potential for performance gains in graph clustering with PIMs, but also some load imbalance, and sensitivity to data set features.

Beyond these results, this project offered a window into the impact of architectural features on performance and suggested architectural improvements, as follows:

- The processor must incorporate a simple load/store unit to overlap data movement with computation. While the PIM memory was just 3 cycles away on our implementation, sometimes this reduced our performance gains, and looking to the future, the latencies of eDRAM are increasing with technology shrinks.

- We encountered limitations in the size of the wide register file in the MutualInformation kernel, and would have benefited from deeper, narrower register file (64x128 bits rather than 32x256 bits).
- There are a number of specialized features we included in our implementation that limited clock speed and were rarely if ever used, for example, the leftmost and rightmost participation operations.
- Additional research is needed in the relationship between on-chip and off-chip scalable networks for PIMs

In conclusion, this was a complex and and interesting project requiring a variety of expertise: link discovery algorithms and general parallel algorithms, compilers, optimization, performance tuning and low-level engineering. It was only possible because we leveraged several million dollars in prior DOD funding (hardware, compilers, and algorithms). However, the short time line of the project was challenging, particularly given that the broad number of skills required that a large number of people be involved – the project supported the equivalent of about two full-time researchers, but spread across eight people. Given the complexity of debugging on an experimental system, a 15-month timeline proved too short to iterate on alternative implementations or complete the analysis of the performance we obtained. Tuned code is even harder to debug, as it uses special instructions, compiler-optimized code and parallel constructs.

Looking to the future, there is much to be done in all the areas supported by this project. In addition to the architectural advances described above, there is a need to develop parallel algorithms for link discovery and identify the appropriate algorithms and programming models to manage communication in graph data structures. We are interested in investigating the transactional memory as a strategy for programming link discovery codes, as we are performing a similar investigation under the DARPA Architectures for Cognitive Information Processing (ACIP) program. Other compiler research could benefit link discovery codes. Interestingly, for both kernels, there was SIMD parallelism that could be exploited automatically by our compiler, and SIMD functional units are becoming common features in not only multimedia extensions but also graphics and other high-performance domain-specific processors such as, for example, the IBM Cell processor [Kahle 2005]. Also, our compiler's approach to compiler-controlled caching of data will be useful in architectures such as Cell, CPUTech, GPUs and others that require data-movement in software-controlled storage.

7. Outreach

During this short project, there were several activities that disseminated information about the work supported by this project. Several participants presented a poster and a demonstration board from the PIM testbeds at the USC research booth at the Supercomputing 2004 conference. A press release on the StreamAdd results came out concurrently with the conference. A research poster on this work appeared at Supercomputing 2005 poster exhibit [Barrett et al 2005] and again at the USC booth. After the project end but at the time of this report, two additional publications on this work have been accepted, a summary of this report [Adibi et al 2006] and a journal article on the compiler technology [Shin et al 2006].

Mary Hall participated in the AFRL Workshop on Research Directions in Architectures and Systems for Cognitive Processing, July 14-15, 2005, at Cornell University (http://csl.cornell.edu/wcas/), and gave a presentation in the Architectures and Systems for Cognitive Processing working group.

A poster on this work was presented at the ISI Industrial Affiliates meeting in June, 2005. Jaewook Shin, a PhD student supported on this project, who developed the frontend compiler, completed his dissertation in May 2005 and presented this work at several universities, government laboratories, and computer companies. He is currently a postdoctoral scientist at Argonne National Laboratories.

In August, 2005, several participants met with representatives from CPU Technology Inc. to discuss the potential for future collaboration.

References

[Adibi et al. 2004a] J. Adibi, H. Chalupsky, E. Melz and A. Valente. The KOJAK Group Finder: Connecting the Dots via Integrated Knowledge-Based and Statistical Reasoning. In Proceedings of the Sixteenth Innovative Applications of Artificial Intelligence Conference (IAAI-04), 2004.

[Adibi et al. 2004b] Adibi, J., Valente, A., Chalupsky, H. & Melz, E. (2004). Group detection via a mutual information model.

[Adibi et al 2006] Adibi, J., Barrett, T., Bhatt, S., Chalupsky, H., Chame, J., Hall, M. "Processing in Memory Technology for Knowledge Discovery Algorithms," Proceedings of the Second International Workshop on Data Management on New Hardware (DaMoN 2006) June 25, 2006, Chicago, Illinois, USA.

[Barrett et al., 2005] T. Barrett, S. Bhatt, J. Chame, J. Draper, "PIMs in Action, Delivering Memory Bandwidth," Poster presentation at the International Conference on Supercomputing, Nov. 2005.

[DIS 2000] Data Intensive Systems Benchmark Suite. http://www.aaec.com/projectweb/dis/ .

[Draper, 2002] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, "The Architecture of the DIVA Processing-In-Memory Chip," In Proceedings of the International Conference on Supercomputing, June, 2002.

[Hall et. al., 1999] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, J. Shin, J. Park, "Mapping Irregular Computations to DIVA, a Data-Intensive Architecture," In Proceedings of the International Conference on Supercomputing, Nov. 1999.

[Hall & Steele, 2000] M.W. Hall and C. Steele, "Memory Management in PIM-Based Systems," In Proceedings of the Workshop on Intelligent Memory Systems, held in conjunction with Architectural Support for Programming Languages and Operating Systems, Boston, MA, Nov. 2000.

[HPC 2005] HPC Challenge Benchmarks. http://icl.cs.utk.edu/hpcc/ .

[Kahle 2005] The Cell Processor Architecture. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, 2005.

[Lin & Chalupsky 2003a] Lin, S. & Chalupsky, H. Using unsupervised link discovery methods to find interesting facts and connections in a bibliography dataset. SIGKDD Explorations, 5(2): 173-178, December 2003

[Lin & Chalupsky 2003b] S. Lin and H. Chalupsky. Unsupervised Link Discovery in Multi-relational Data via Rarity Analysis. In Proceedings of the Third IEEE International Conference on Data Mining (ICDM '03). 2003

[Senator 2002] Senator, T. (2002). Evidence Extraction and Link Discovery, DARPA Tech 2002.

[Shin2002a] J. Shin, J. Chame and M. W. Hall, "Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures." In Proceedings of the Parallel Architectures and Compilation Techniques Conference, Sept. 2002.

Shin2002b] J. Shin, J. Chame and M. W. Hall, "A Compiler Algorithm for Exploiting Page-Mode Memory Accesses in Embedded-DRAM Devices," In Proceedings of the Fourth Workshop on Media and Stream Processors Workshop, held in conjunction with MICRO '02, November, 2002.

[Shin2003] J. Shin, J. Chame and M. W. Hall, "Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures," Distinguished paper selected from PACT '02, Journal of Instruction-Level Parallelism, 2003.

[Shin 2004] J. Shin, M. Hall and J. Chame, ``Evaluating Compiler Technology for Control-Flow Optimizations for Multimedia Extension Architectures,". In Proceedings of the Media-Specific Processors Workshop, held in conjunction with MICRO '04, December, 2004.

[Shin 2005] J. Shin, M. Hall and J. Chame, ``Superword-Level Parallelism in the Presence of Control Flow," In Proceedings of the Conference on Code Generation and Optimization, March, 2005.

[Shin et al 2006] J. Shin, M. Hall and J. Chame, "'Evaluating Compiler Technology for Control-Flow Optimizations for Multimedia Extension Architectures," Invited paper to appear in the International Journal of Embedded Systems, 2006.

[STELLA 2003] http://www.isi.edu/isd/LOOM/Stella/ .

Table of Acronyms

Acronym	**Meaning**
ACIP	Architectures for Cognitive Information Processing
AFRL	Air Force Research Laboratory
ARDA	Advanced Research and Development Activity
DIMM	Dual Inline Memory Module
DIVA	Data Intensive Architecture
DRAM	Dynamic Random Access Memory
EAGLE	Evidence Assessment, Grouping, Linking, and Evaluation
GUPS	Giga Updates Per Second
IOR	In-Out Ratio
IT2	Itanium-2
LD	Link Discovery
MCHIP	Monarch Cognitive Heterogeneous Information Processor
MI	Mutual Information
MPI	Message Passing Interface
PBUF	Parcel Buffer
PIM	Processing in Memory
PIRC	Pim Routing Component
SIMD	Single Instruction Multiple Data
SPL	Superword-Level Parallelism
SUIF	Stanford University Infrastructure